

# Vorlesung Compilerbau (SoSe 2018)

## Teil 11: Laufzeitumgebung(en)



### Vorlesung Compilerbau: Laufzeitumgebung(en)

Vorlesung des BA-Studiums  
Prof. Johann Christoph Freytag, Ph.D.  
Institut für Informatik, Humboldt-Universität zu Berlin  
SoSe 2018

© Prof. J.C. Freytag, Ph.D.

Bitte Handys leise schalten....

11.1

## Überblick

Zugriff auf Variablen zur Laufzeit  
(Beziehung zwischen Namen und Datenobjekten)

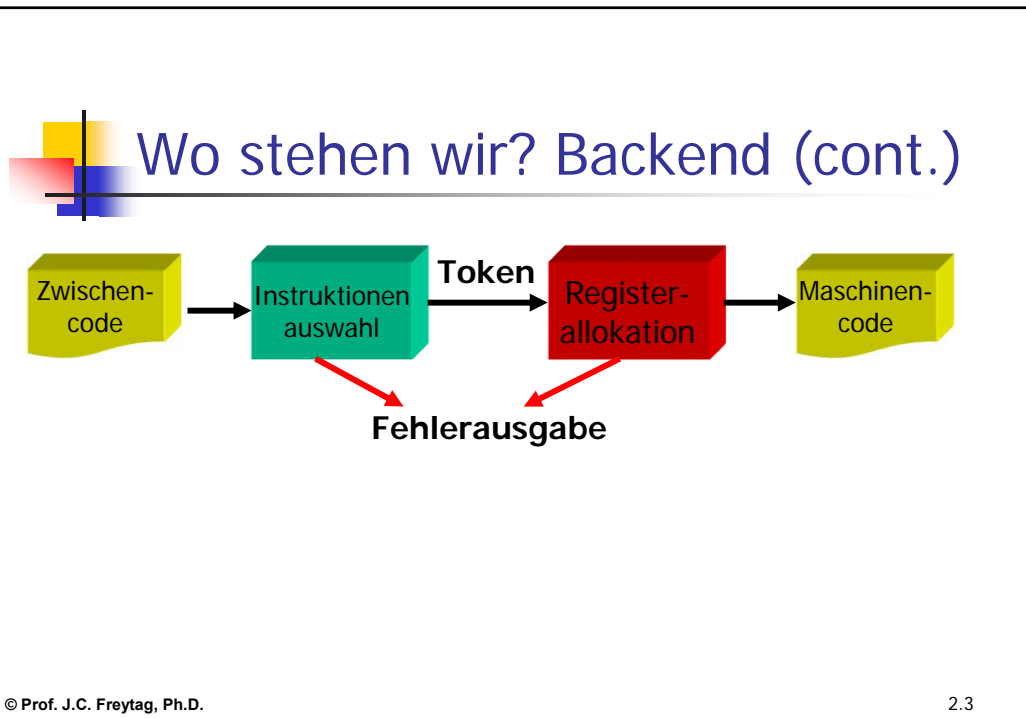
- Speicherbehandlung und -verwaltung
- Aktivierungssegmente
- Zugriffsverweis (Access-Link)
- Display

© Prof. J.C. Freytag, Ph.D.

11.2

# Vorlesung Compilerbau (SoSe 2018)

## Teil 11: Laufzeitumgebung(en)



### Quellsprache

- sei hier zunächst der Einfachheit halber prozedural (Pascal, Fortran, Lisp, ...)
- problematischer sind beispielsweise Sprachen mit Prozesskonzepten (Ada, ...)

© Prof. J.C. Freytag, Ph.D. 11.4

# Prozedurkonzept

## Separierung für die Übersetzung

- Entwurf und Realisierung großer Programme
- hält Übersetzungszeiten in Grenzen
- erfordert unabhängige Prozeduren

## Die »Verbindungs«-Konvention (engl. linkage)

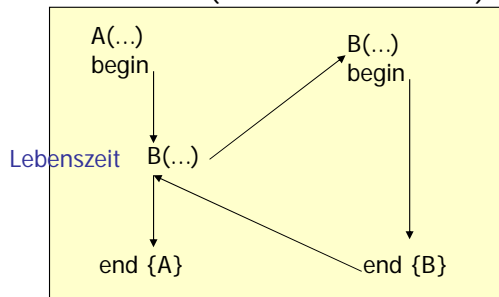
- stellt sicher, dass Prozeduren zur Laufzeit eine korrekte Laufzeitumgebung erhalten und nach Beendigung eine korrekte Laufzeitumgebung für die aufrufende Prozedur wiederherstellen
- Verbindungen (Code) werden zur Laufzeit ausgeführt
- Code zur Herstellung einer Verbindung wird bereits zur Übersetzungszeit generiert

## Ausführungsbäume

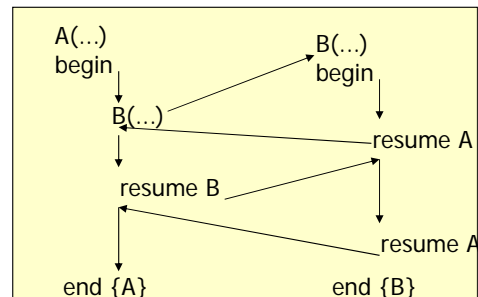
## Annahmen zum Kontrollfluss

- sequentiell (schrittweise Abarbeitung)
- **Prozedurausführung** beginnt am Anfang des Prozedurkörpers und führt hinter den Prozeduraufruf zurück

## Prozeduren (ohne Ausnahmefehler)



### Alternativ: Co-Routinen



# Vorlesung Compilerbau (SoSe 2018)

## Teil 11: Laufzeitumgebung(en)

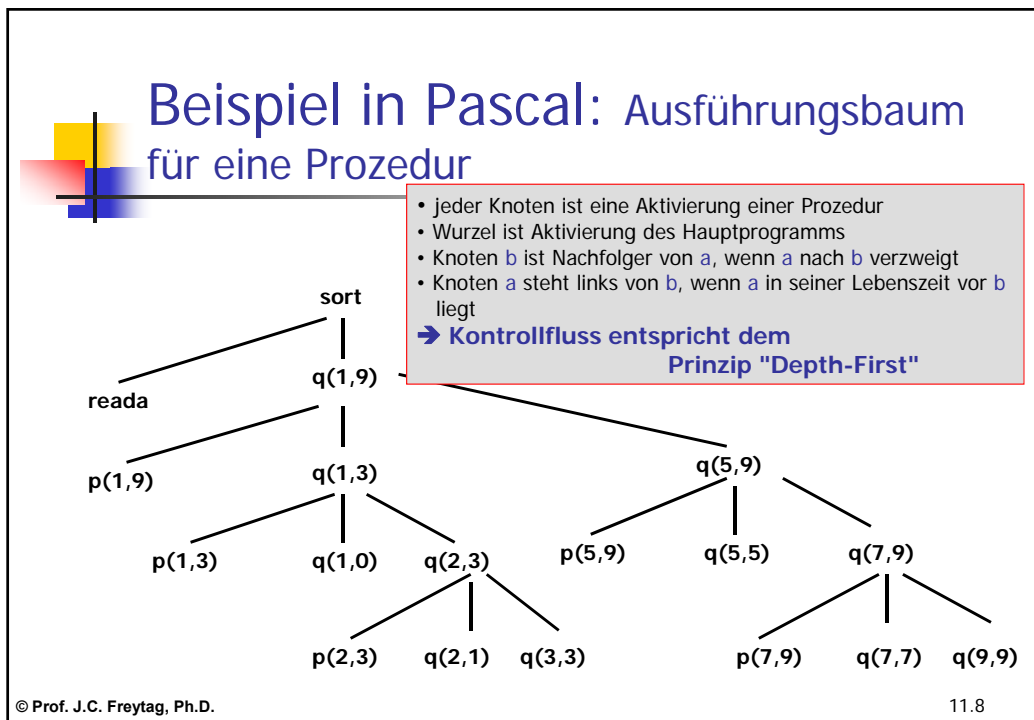
### Beispiel in Pascal: Einlesen u. Sortieren von integer-Zahlen

```
program sort (input, output);  
  var a : array [0..10] of integer;  
      x : integer;  
  
  procedure readarray;  
    var i : integer;  
    begin ... a[i] ... end {readarray};  
  
  function partition (y, z : integer): integer;  
    var i, j, x, v : integer;  
    begin ... end {partition};  
  
  procedure quicksort (m, n : integer);  
    var i : integer;  
    begin {quicksort}  
      if (n > m) then begin  
        i := partition(m,n);  
        quicksort(m, i-1);  
        quicksort (i+1;n);  
      end  
    end {quicksort};  
  
  begin {sort}  
    readarray;  
    quicksort(1,9)  
  end {sort};
```

**Prozedur-Verallgemeinerung**

- Hauptprogramm
- Funktionen
- Prozeduren

© Prof. J.C. Freytag, Ph.D. 11.7



# Vorlesung Compilerbau (SoSe 2018)

## Teil 11: Laufzeitumgebung(en)

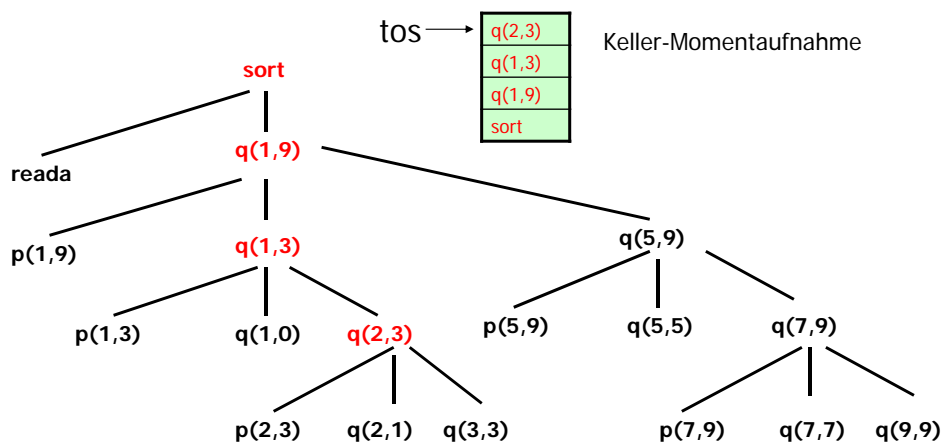
### „Steuerungskeller“: Aufgaben

#### Keller zur Ablaufsteuerung

- merkt sich (noch) »lebende« Prozeduraktivierungen
- Knoten wird auf den Keller abgelegt, wenn Aktivierung beginnt bzw. wird bei Beendigung der Prozedur entnommen
- Kellerinhalt ist der Pfad des Aktivierungsbaums

#### Keller aber auch zur Speicherzuweisung

### Beispiel in Pascal: Aufbau des Steuerungskellers



# Vorlesung Compilerbau (SoSe 2018)

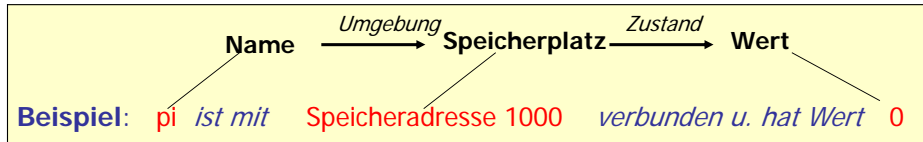
## Teil 11: Laufzeitumgebung(en)



### Bindungen von Namen

#### Problemstellung

- Bindung eines Namens **x** an eine Speicherplatz **s**



- Bindung findet zur Laufzeit statt  
(dynamisches Gegenstück zur Deklaration eines Namens während der Übersetzungszeit)

statische Notation	dynamische Notation
Definition einer Prozedur	Aktivierung einer Prozedur
Deklaration eines Namens	Bindung dieses Namens
Gültigkeitsbereich einer Deklaration	Lebenszeit der Bindung



### Bindungen von Namen

#### beeinflusst durch Antwort auf wichtige Fragen

- Dürfen Prozeduren rekursiv sein?
- Darf eine Prozedur auf nicht-lokale Namen zugreifen?
- Welche Art der Parameterübergabe ist zugelassen?
- Dürfen Funktionen als Parameter übergeben werden?
- Darf eine Funktion als Ergebnis zurückgegeben werden?
- Ist dynamische Speicherallokation erlaubt?
- Muss Speicher explizit/implizit freigegeben werden?

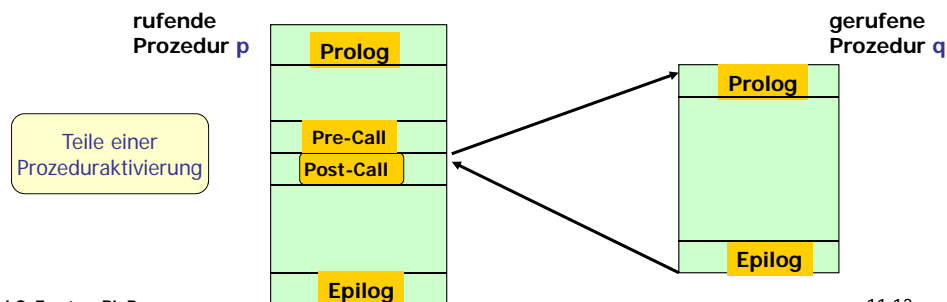
# Vorlesung Compilerbau (SoSe 2018)

## Teil 11: Laufzeitumgebung(en)

### Aktivierung von Prozeduren

Wichtigste Aspekte einer Prozeduraktivierung

- bei Eintritt in  $p$ , etabliere Laufzeitumgebung von  $p$
- bei Aufruf von  $q$ , erhalte Laufzeitumgebung von  $p$
- beim Verlassen von  $q$ , lösche Laufzeitumgebung von  $q$
- während der Ausführung von  $q$ , stelle den Zugriff auf alle gültigen Namen sicher



© Prof. J.C. Freytag, Ph.D.

11.13

### Aktivierung von Prozeduren (Forts.)

Wie ist die Aktivierung zu organisieren?

Benötige Datenstruktur, die

- die Übergabe von wichtigen Daten (hin und zurück) organisiert,
- flexibel und
- platzsparend ist

gängige Lösung

- Nutzung eines Laufzeitkellers
  - beliebige Aufrufsequenz und -tiefe
- **Aktivierungssegmente** oder auch Rahmen (engl. »Frames«)
  - werden mit jedem Aufruf erzeugt und auf dem Keller gespeichert
  - dienen zur Realisierung des Prologs und Epilogs

© Prof. J.C. Freytag, Ph.D.

11.14

# Vorlesung Compilerbau (SoSe 2018)

## Teil 11: Laufzeitumgebung(en)



### Speicherorganisation zur Laufzeit (Forts.)

#### Fragen

- Wo werden lokale Variablen gespeichert?
- Wann können sie auf dem Keller abgelegt werden?  
Schlüsselfrage: Lebenszeit eines lokalen Namens/lokalen Speicherzelle

#### zwei Aspekte

1. weiterzugebende Daten:
    - Zugriff der gerufenen Prozedur auf Daten der rufenden Prozedur
    - Wichtiger Unterschied:
      - **dynamisches Scoping** (Gültigkeitsbereich)
      - **lexikalisches Scoping**
  2. zurückzugebende Daten
    - Können Referenzen auf Daten der gerufenen Prozedur zurückgegeben werden?
    - Funktionen die Funktionen als Ergebnis zurückgeben
- bei ausschließlich weiterzugebenden Daten kann der Compiler die Frames auf dem Laufzeitkeller anlegen (allokieren)



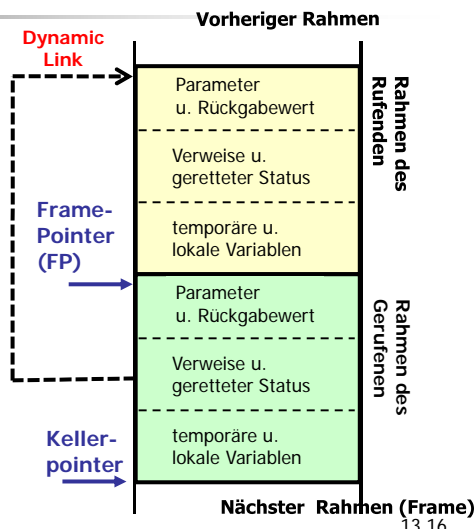
### Aktivierung von Prozeduren (Forts.)

#### Annahme:

- jeder Prozeduraufruf (»Aktivierung«) ist mit einem **Aktivierungssegment** (**Rahmen**, engl. **frame**) zur Laufzeit assoziiert
- dynamisch allokierte Segmente können dynamisch erweitert werden
- aktuelle Parameterwerte werden bei Aufruf im Rahmen gespeichert
- lokale Variablen werden im Rahmen gespeichert

#### zur Realisierung:

- **Frame-Pointer**: zeigt den Beginn des momentan »aktiven« Rahmens an
- **Kellerpointer**: zeigt den momentanen **tos** an





# Vorlesung Compilerbau (SoSe 2018)

## Teil 11: Laufzeitumgebung(en)



### Aufbau von Aktivierungssegmenten (Frames)

- temporäre Daten  
Werte von Ausdrücken
- lokale Daten
- geretteter „Maschinenzustand“  
Zustand vor Aufruf:  
IP (Instruction Pointer),  
Werte der Register (die bei Rückkehr wieder benötigt werden)
- optionaler Zugriffsverweis  
Verweis auf (nichtlokale) Daten anderer Aktivierungssegmente (static link)
- optionaler Steuerungsverweis  
Verweis zum Aktivierungssegment der rufenden Prozedur (dynamic link)
- aktuelle Parameter  
Parameterfeld wird von rufender und gerufener Prozedur benutzt
- Rückgabewert  
Parameterfeld wird von rufender und gerufener Prozedur benutzt



### Aktivierung von Prozeduren (Forts.)

geteilte Verantwortung zwischen rufender und gerufener Prozedur

	rufende Prozedur	gerufene Prozedur
Aufruf	Pre-Call	Prolog
Rücksprung	Post-Call	Epilog

# Vorlesung Compilerbau (SoSe 2018)

## Teil 11: Laufzeitumgebung(en)

### Aktivierung von Prozeduren (Forts.)

geteilte Verantwortung zwischen rufender und gerufener Prozedur

	rufende Prozedur	gerufene Prozedur
Aufruf	<b>Pre-Call</b> 1. Allokieren Basis-Frame 2. Berechne und speichere Parameter 3. Springe zur gerufenen Prozedur	<b>Prolog</b>
Rücksprung	<b>Post-Call</b>	<b>Epilog</b>

© Prof. J.C. Freytag, Ph.D.

11.19

### Aktivierung von Prozeduren (Forts.)

geteilte Verantwortung zwischen rufender und gerufener Prozedur

	rufende Prozedur	gerufene Prozedur
Aufruf	<b>Pre-Call</b> 1. Allokieren Basis-Frame 2. Berechne und speichere Parameter 3. Springe zur gerufenen Prozedur	<b>Prolog</b> 1. Sichere Register, Zustand, Rücksprung 2. Speichere Frame-Pointer (FP) 3. Setze neuen FP 4. Speichere statische Links /* nicht in c */ 5. Erweitere Basis-Frame für lokale Daten (Größe ~ Typ) 6. Initialisiere lokale Daten /* nicht in c */ 7. Springe zum eigentlichen Code der Prozedur
Rücksprung	<b>Post-Call</b>	<b>Epilog</b>

Weil keine  
Schachtelung

C initialisiert  
nicht

© Prof. J.C. Freytag, Ph.D.

11.20

# Vorlesung Compilerbau (SoSe 2018)

## Teil 11: Laufzeitumgebung(en)

### Aktivierung von Prozeduren (Forts.)

geteilte Verantwortung zwischen rufender und gerufener Prozedur

	rufende Prozedur	gerufene Prozedur
<b>Aufruf</b>	<b>Pre-Call</b>	<b>Prolog</b>
	<ol style="list-style-type: none"> <li>1. Allokiere Basis-Frame</li> <li>2. Berechne und speichere Parameter</li> <li>3. Springe zur gerufenen Prozedur</li> </ol>	<ol style="list-style-type: none"> <li>1. Sichere Register, Zustand, Rücksprungadresse</li> <li>2. Speichere Frame-Pointer (FP)</li> <li>3. Setze neuen FP</li> <li>4. Speichere statische Links /* nicht in c */</li> <li>5. Erweitere Basis-Frame für lokale Daten (Größe ~ Typ)</li> <li>6. Initialisiere lokale Daten /* nicht in c */</li> <li>7. Springe zum eigentlichen Code der Prozedur</li> </ol>
<b>Rücksprung</b>	<b>Post-Call</b>	<b>Epilog</b>
		<ol style="list-style-type: none"> <li>1. Speichere Ergebniswerte im Frame</li> <li>2. Stelle alten Zustand wieder her</li> <li>3. Reduziere Rahmen auf Basisrahmen</li> <li>4. Stelle den FP der aufrufenden Prozedur wieder her</li> <li>5. Springe zur Rücksprungadresse</li> </ol>

© Prof. J.C. Freytag, Ph.D.

11.21

### Aktivierung von Prozeduren (Forts.)

geteilte Verantwortung zwischen rufender und gerufener Prozedur

	rufende Prozedur	gerufene Prozedur
<b>Aufruf</b>	<b>Pre-Call</b>	<b>Prolog</b>
	<ol style="list-style-type: none"> <li>1. Allokiere Basis-Frame</li> <li>2. Berechne und speichere Parameter</li> <li>3. Springe zur gerufenen Prozedur</li> </ol>	<ol style="list-style-type: none"> <li>1. Sichere Register, Zustand, Rücksprungadresse</li> <li>2. Speichere Frame-Pointer (FP)</li> <li>3. Setze neuen FP</li> <li>4. Speichere statische Links /* nicht in c */</li> <li>5. Erweitere Basis-Frame für lokale Daten (Größe ~ Typ)</li> <li>6. Initialisiere lokale Daten /* nicht in c */</li> <li>7. Springe zum eigentlichen Code der Prozedur</li> </ol>
<b>Rücksprung</b>	<b>Post-Call</b>	<b>Epilog</b>
	<ol style="list-style-type: none"> <li>1. Kopiere Ergebniswerte</li> <li>2. Deallokiere Basisrahmen der aufgerufenen Prozedur</li> </ol>	<ol style="list-style-type: none"> <li>1. Speichere Ergebniswerte im Frame</li> <li>2. Stelle alten Zustand wieder her</li> <li>3. Reduziere Rahmen auf Basisrahmen</li> <li>4. Stelle den FP der aufrufenden Prozedur wieder her</li> <li>5. Springe zur Rücksprungadresse</li> </ol>

© Prof. J.C. Freytag, Ph.D.

11.22

# Vorlesung Compilerbau (SoSe 2018)

## Teil 11: Laufzeitumgebung(en)

### Speicherorganisation zur Laufzeit

#### Konventionen für die Speicherplatzaufteilung

1. Speicherplatz für **Programcode**
  - feste Länge
  - deshalb statische Allokation
2. Speicherplatz für **Daten**
  - Daten fester Länge können bereits **statisch** allokiert werden  
**Vorteil:** Compiler kann Adressen schon fest in den Code eintragen (bei lokalen Variablen nur relativ zum FP möglich)
  - Daten variabler Länge müssen **dynamisch** allokiert werden
3. (möglicherweise) **Steuerungskeller** (für Pascal, C, ...)
  - beinhaltet Teil zur Verwaltung von Prozeduraktivierungen (**Aktivierungssegmente**)
  - inklusive die Rücksprungadresse
  - spezielle Maschinenbefehle für Stack-Manipulationen (Sparc)

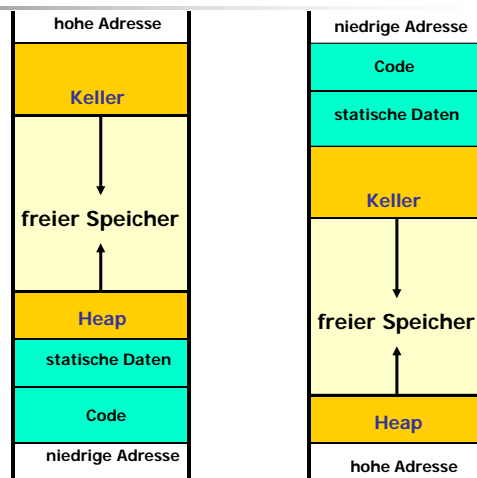
### Speicherorganisation zur Laufzeit

#### typisches Schema zur Organisation des Speichers

- gibt sowohl dem Keller als auch dem Heap maximale Flexibilität und Freiheiten
- Code und statische Daten können miteinander »verquickt« werden

#### Achtung:

- Keller wächst hier von hohen auf kleinere Adresswerte
  - Heap von kleineren auf größere Adresswerte
- oder umgekehrt**



# Vorlesung Compilerbau (SoSe 2018)

## Teil 11: Laufzeitumgebung(en)



### Speicherklassen

**Jeder** Variablen muss eine *Speicherklasse* zugewiesen werden (Basisadresse)

Speicherklasse: wo ist Speicherplatz zu allokalieren?

- **statische Variablen** (in C static, für Modul zugreifbar)
  - Speicheradressen werden in den Code eingesetzt
  - (meist) werden diese zur Übersetzungszeit allokiert
  - **begrenzt auf Objekte fester Größe**
  - Zugriff wird *mit einem Namensschema kontrolliert* (später)
- **globale Variablen** (für ganzes Programm zugreifbar)
  - fast identisch zur statischen Variablen
  - Namensschema garantiert universellen Zugriff

**Achtung:**

Der **Linker** (der einzelne Module zu einem Programm zusammenführt) muss Duplikate der statischen Variablen handhaben können (z.B.: Modul-Name/Variablen-Name)



### Speicherklassen (Forts.)

- **lokale Variablen einer Prozedur**
  - werden auf dem Keller gespeichert, falls
    - die Größe fest ist
    - die Lebenszeit mit der Lebenszeit der Prozedur übereinstimmt und
    - der Wert nicht aufgehoben werden muss
  - müssen anders behandelt werden, wenn sie dynamisch allokiert wurden
  - Call-by-reference:
    - Zeiger (Pointer) erzeugen Werte mit nicht-lokalen Lebenszeit

# Vorlesung Compilerbau (SoSe 2018)

## Teil 11: Laufzeitumgebung(en)

### lokale Objekte variabler Länge

- eine Prozedur p habe 3 lokale Felder (Arrays) A, B, C
- Speicher für die Felder ist *nicht* Bestandteil des Aktivierungssegmentes von p dafür aber Zeiger (die relativen Adressen dieser Zeiger sind bereits zur Compile-Zeit bekannt!)  
→ Zielcode kann deshalb auf Feldelemente zugreifen
- das Aktivierungselement der gerufenen Prozedur liegt nach dem Segment für die Felder

Vorheriger Rahmen

Rahmen des Rufenden

Stack oder Heap

Rahmen des Gerufenen

Nächster Rahmen

© Prof. J.C. Freytag, Ph.D. 11.27

### Hängende (dangling) Referenzen

- ...sind Referenzen auf einen Speicherplatz, der bereits freigegeben ist
- Verwendung solcher Referenzen ist Ursache für mysteriöse Fehler!!!

```
int main () {  
    int *p;  
    p = dangle();  
}  
int *dangle() {  
    int i = 23;  
    return &i;  
}
```

- p zeigt nach Verlassen von dangle auf einen solchen Speicherplatz

Vorheriger Rahmen

Rahmen des Rufenden

Rahmen des Gerufenen

Nächster Rahmen

© Prof. J.C. Freytag, Ph.D. 11.28

# Vorlesung Compilerbau (SoSe 2018)

## Teil 11: Laufzeitumgebung(en)



### Heap-Zuweisung

- Kellerzuweisungsstrategie für Aktivierungs-segmente kann **nicht** benutzt werden, wenn
  1. Werte lokaler Namen erhalten bleiben müssen
  2. eine gerufene Prozedur die aufrufende Prozedur überlebt (z.B. bei Koroutinen)
- Problem der Heap-Zuweisungsstrategie

Heap besteht (nach einiger Zeit) aus Bereichen, die frei oder in Benutzung sind.



### Zugriff auf nicht-lokale Daten

Wie werden *nicht-lokale Daten* zur Laufzeit gefunden?

- (Compilererzeugter) Code muss diesen Zugriff beinhalten/realisieren
- echt *globale* Variablen
  - Konventionen (Festlegung) im Compiler gibt die Adresse des Speicherplatzes an
  - Initialisierung muss gewährleistet sein (falls gefordert)
- **lexikalische (oder statische) Bindung:**  
Zugriff auf Variablen der statischen Umgebung
  - lexikalisch: Gültigkeitsbereiche werden anhand der Definition im Programm (zur **Übersetzungszeit**) bestimmt
  - realisiert durch (level, offset)-Paare (später)
- **dynamische Bindung** (z.B. Lisp: anwendbare Deklaration wird zur Laufzeit ermittelt): Zugriff auf Variablen des Rufers

# Vorlesung Compilerbau (SoSe 2018)

## Teil 11: Laufzeitumgebung(en)



### Blöcke

in C

{ deklarations statements }

**Merkmal:** geschachtelte Struktur (Block-Level)  
kann durch Kellerspeicher (Aktivierungselement) repräsentiert werden



### Beispiel quicksort

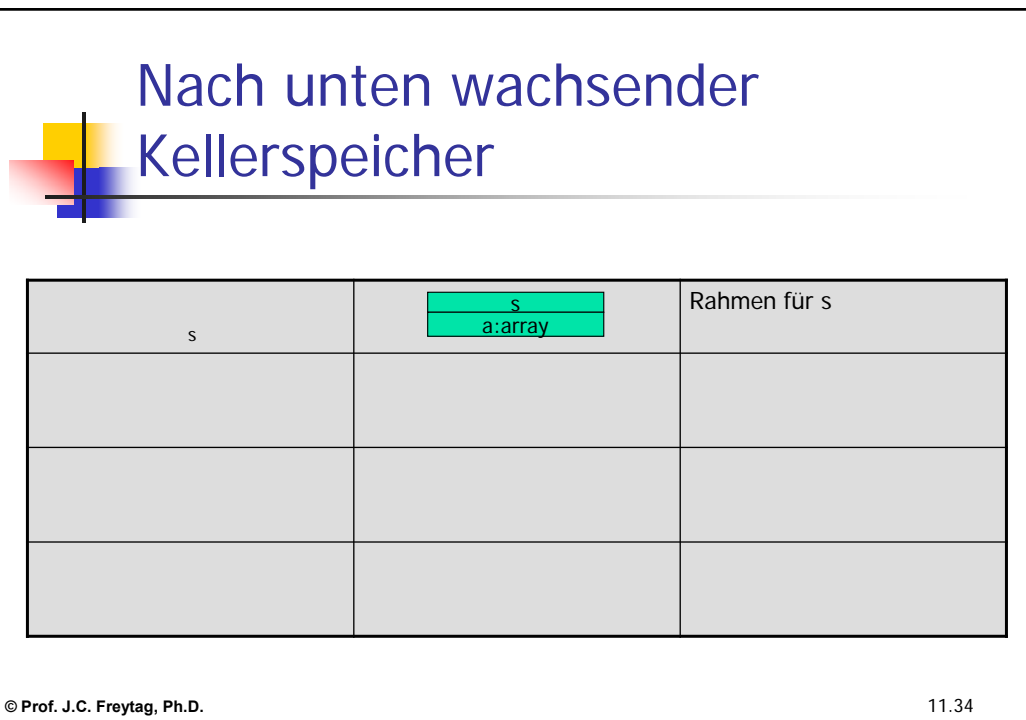
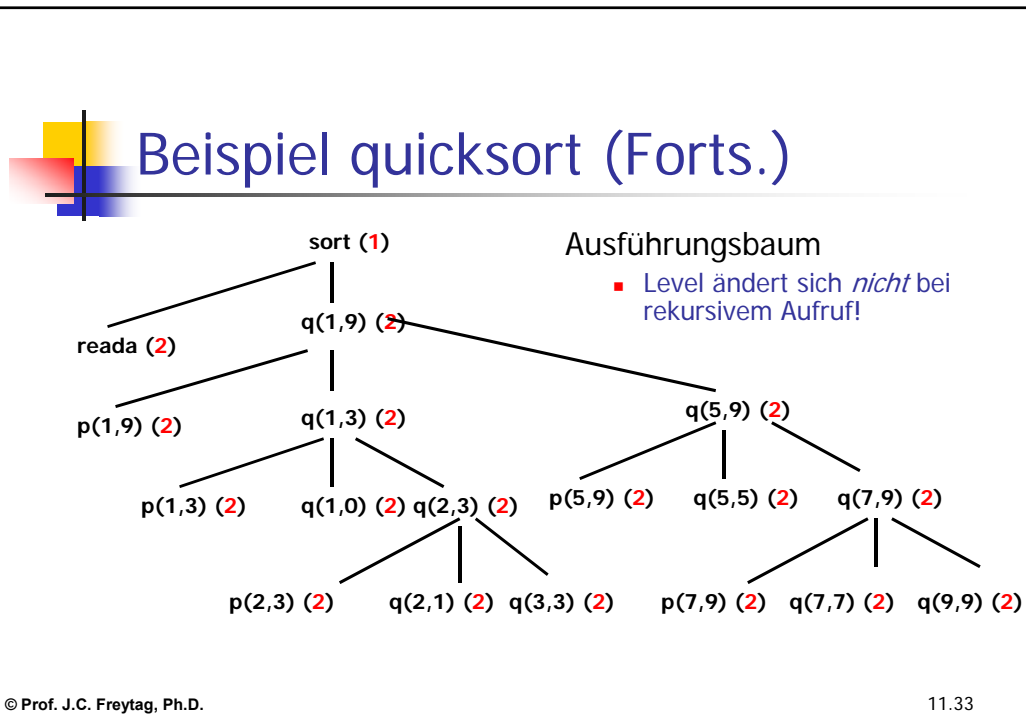
```
program sort (input, output) : { level 1 }  
  var a : array [0..10] of integer;  
  x : integer;  
  
  procedure readarray; { level 2 }  
    var i : integer;  
    begin ..... a[i] ..... end {readarray};  
  
  function partition (y, z : integer): integer;  
    { level 2 }  
    var i, j, x, v : integer;  
    begin ..... end {partition};
```

```
procedure quicksort (m, n : integer) : { level 2 }  
  var i : integer;  
  begin {quicksort}  
    if (n > m) then begin  
      i := partition(m,n);  
      quicksort(m, i-1);  
      quicksort (i+1;n);  
    end  
    end {quicksort};  
  
  begin {sort }  
    readarray;  
    quicksort(1,9)  
  end {sort } .
```



# Vorlesung Compilerbau (SoSe 2018)

## Teil 11: Laufzeitumgebung(en)



# Vorlesung Compilerbau (SoSe 2018)

## Teil 11: Laufzeitumgebung(en)

### Nach unten wachsender Kellerspeicher (cont.)

s	<div> <div>s</div> <div>a:array</div> </div>	Rahmen für s
	<div> <div>s</div> <div>a:array</div> <div>r</div> <div>i:integer</div> </div>	r ist aktiviert

© Prof. J.C. Freytag, Ph.D. 11.35

### Nach unten wachsender Kellerspeicher (cont.)

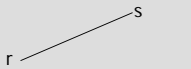
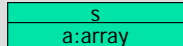

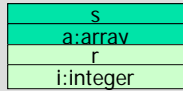
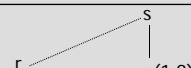
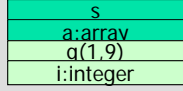
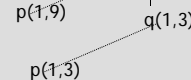
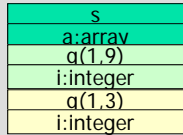
s	<div> <div>s</div> <div>a:array</div> </div>	Rahmen für s
	<div> <div>s</div> <div>a:array</div> <div>r</div> <div>i:integer</div> </div>	r ist aktiviert
	<div> <div>s</div> <div>a:array</div> <div>q(1,9)</div> <div>i:integer</div> </div>	Rahmen von r wurde entfernt u. der für q(1,9) daraufgesetzt

© Prof. J.C. Freytag, Ph.D. 11.36

# Vorlesung Compilerbau (SoSe 2018)

## Teil 11: Laufzeitumgebung(en)

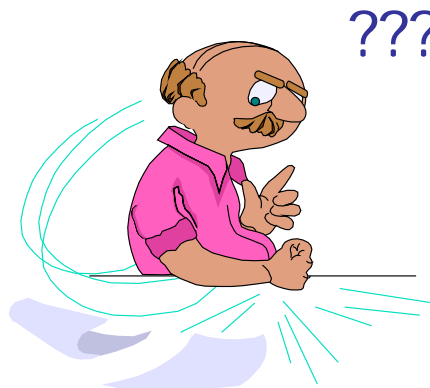
### Nach unten wachsender Kellerspeicher (cont.)

		Rahmen für s
		r ist aktiviert
		Rahmen von r wurde entfernt u. der für q(1,9) daraufgesetzt
		Steuerung ist gerade zu q(1,3) zurückgekehrt

© Prof. J.C. Freytag, Ph.D.

11.37

### Fragen



© Prof. J.C. Freytag, Ph.D.

11.38