Prozeduren

Algorithmen und Datenstrukturen im WS 2005/06

Definition von Prozeduren

Prozeduren und Funktionen

- Definition von Prozeduren und Parametern
  - Prozedur
    - gemeinsam genutzte Anweisungsfolge
      - andere Rechnung ausführen
      - vor jeder Ausführung aktuelle Werte bestimmen
        - Parameter (parameter)
        - aktuelle Parameter
        - formale Parameter

Prozeduren und Funktionen

- Definition von Prozeduren und Parametern
  - Definition von Prozeduren
    - Schlüsselwort (function, procedure, proc, subroutine usw.)
      - in C, C++, Java Funktion ausschließlich durch '(' hinter Namen
    - Rückgabetyp (bei Funktionen)
      - In Pascal (Modula 2, Ada usw.) hinter der Parameterliste.
    - Name der Prozedur
      - selbsterklärend!
        Nicht **`Proc1`**, **`Proc2`**, ...
        sondern **`istFertig`**, **`DruckeZahl`**, **`getInt`**;
      - Name ist 'Adresse' der Prozedur (Marke in 3AA)

```
cout << FunktionsName;
cout << FunktionsName();
```

Prozeduren und Funktionen

- Definition von Prozeduren und Parametern
  - Prozedurtypen
    - Name kann nacheinander verschiedene Prozeduren kennzeichnen
    - Anzahl/Typ der Parameter jeder Prozedur gleich
      - In C: Anzahl/Typ kein Attribut der Prozedur, also beliebig
    - Umgebung einer Prozedur
    - in objektorientierten Sprachen verschiedene Prozeduren
  - Nach Prozedurnamen Liste der Parameter (Parameterliste)
  - Rumpf (body)
    - lokale Deklarationen
    - eigentliche Anweisungsfolge
    - Umgebung
  - Dies ist übliche Reihenfolge
    - Pascal: Typbezeichner hinter der Bezeichnerliste

Prozeduren und Funktionen

- Definition von Prozeduren und Parametern
  - Definition gleicher syntaktischer Aufbau wie Gebrauch
    - Parameter in Listen hinter Prozedurnamen
    - Prozedurname überdefinierter Operatorname
      - in Ada für Überdefinition der Multiplikationsoperation
      - `function "*"(Vector1, Vector2: VectorTyp) return real;`
      - `Product := StartVector * EndVector;`
  - Methoden in Java™
    - Funktionen (*method*) nur innerhalb einer Klasse definiert
    - Syntax wie bei C und C++
    - `<type> <functionName> ( <parameterListe> ) { <rumpf> };`
      - `int wurzel(int parameter) {...}`
      - `ergebnis = wurzel(parameter); // Aufruf einer Funktion`
      - `Zufall = BerechneNeuenZufallswert();`
    - Compiler erkennt eine Funktion an öffnender Klammer "(".
      - gilt auch bei Aufruf einer Funktion

Prozeduren und Funktionen

- Parameterliste
  - formale Prozedurparameter
    - bei Deklaration als 'Platzhalter' für Werte/Objekte
    - meist hinter Prozedurname definiert
  - aktuelle Parameter
    - bei Aufruf der Prozedur
      - müssen mit Werten versehen werden
    - Infix- oder Präefixnotation
      - `a+b; funktionsname(a,b);`
    - Parameter stellen Verbindung zwischen Prozedur und aufrufender Umgebung dar
    - bei Aufruf
      - Werte für aktueller Parameter berechnet
      - an formale Parametern zugewiesen
      - Parameter können verschiedene Typen haben
      - Übergabemechanismen

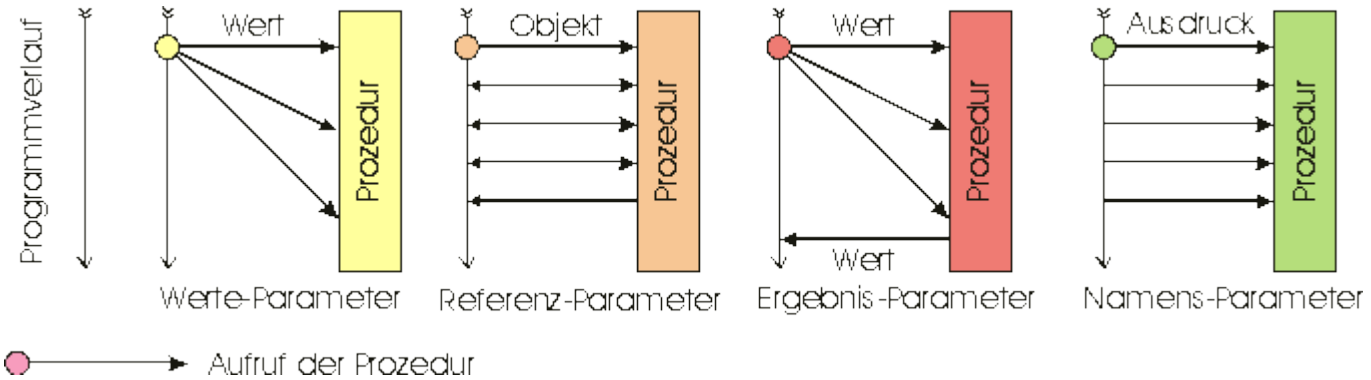
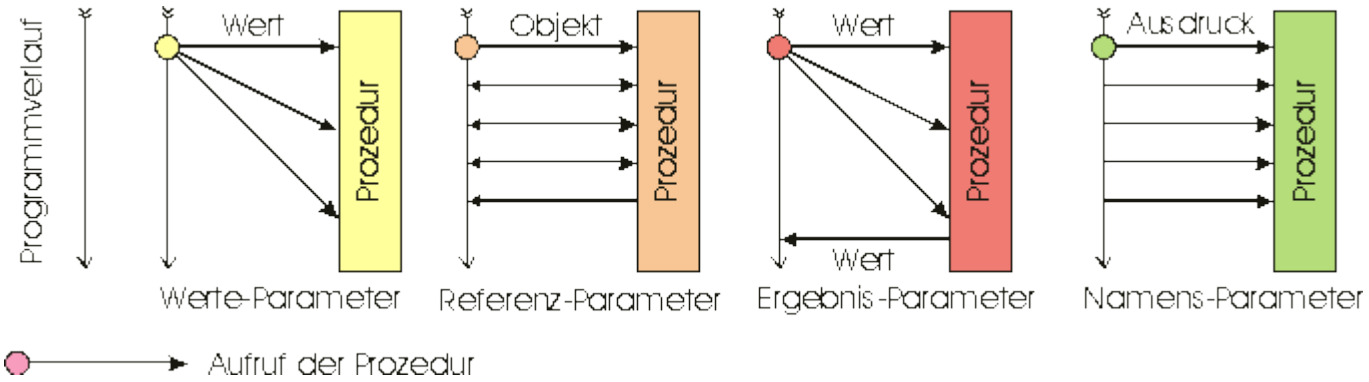
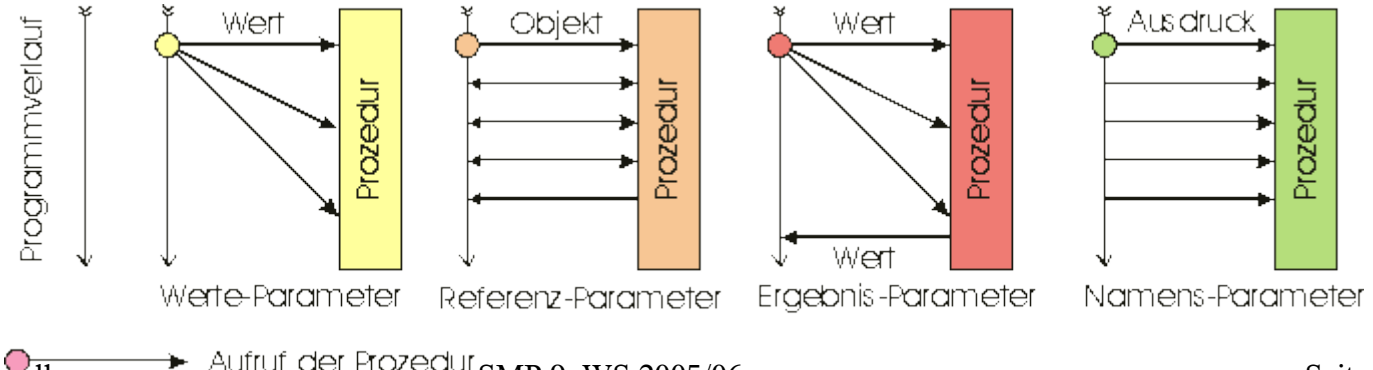
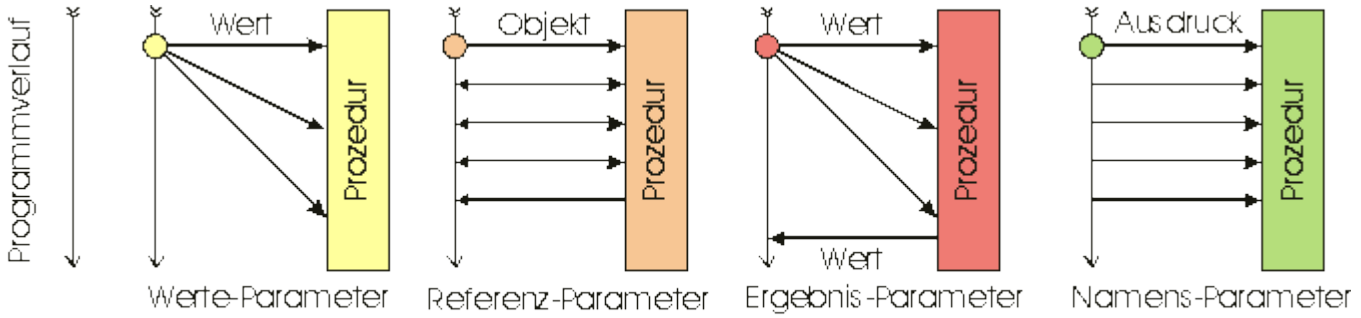
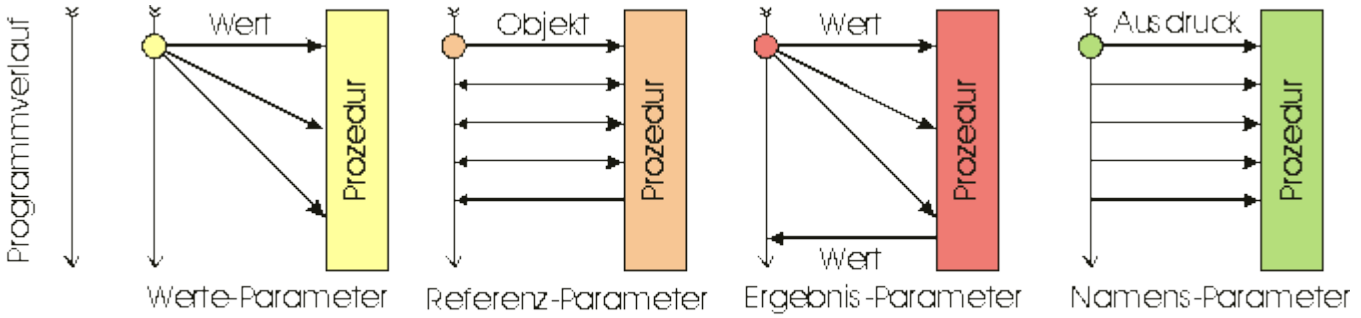
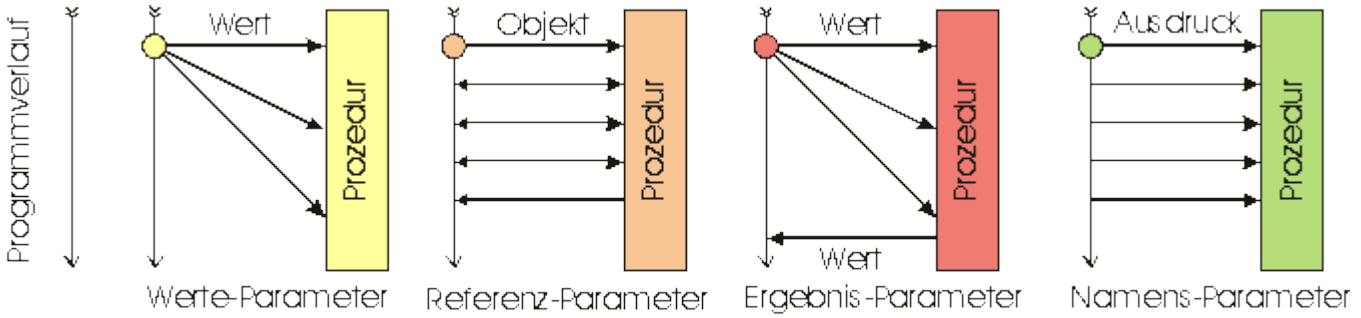
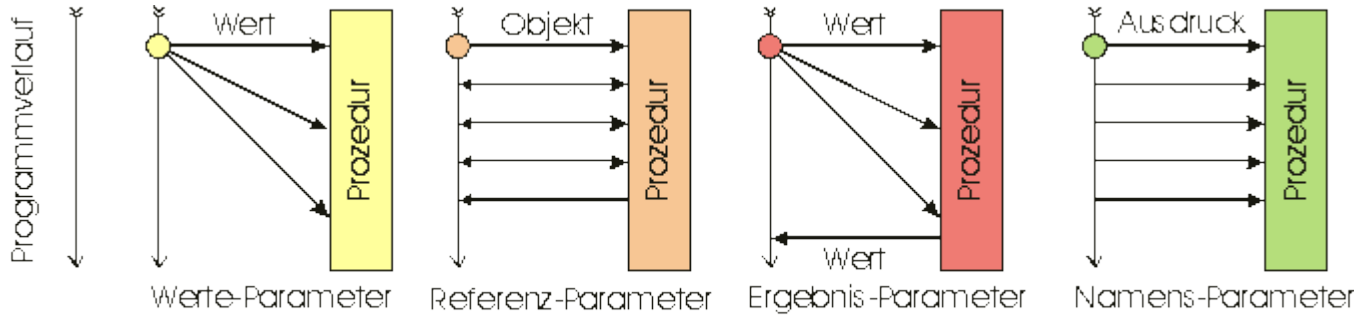
Prozeduren und Funktionen

- Parameterliste
  - Typkompabilität
    - an aufrufender Stelle
      - Werte, die in Typ des jeweiligen formalen Parameters umgesetzt werden können
      - in meisten modernen höheren Sprachen verlangt
        - Pascal, Modula-2 oder Ada
      - C keine Prüfung
      - C++ strenge Typprüfung (mit Überladung)
    - Wertesemantik
      - Ganzzahlausdruck als aktueller Parameter in Gleitpunktausdruck, wenn entsprechender formaler Parameter vom Typ real
    - Zuordnung zwischen formalen und aktuellen Parametern
      - meistens über Reihenfolge in der Parameterliste
      - In Ada beliebige Reihenfolge bei Aufruf
        - `AddModulo100(Summe => AktuelleSumme; Summand => 10);`

Prozeduren und Funktionen

- Parameterliste
  - Parameter mit impliziten Werten
    - **procedure**
      AddModulo100 (integer constant Summand := 100;
      reference integer Summe);
      AddModulo100(Summe => AktuelleSumme);
    - erhöht Wert von **AktuelleSumme** um 1
  - erhöht Flexibilität einer Sprache
  - schränkt Kontrollmöglichkeit durch Compiler ein
    - Vergessen eines Parameters kann nicht mehr entdeckt werden

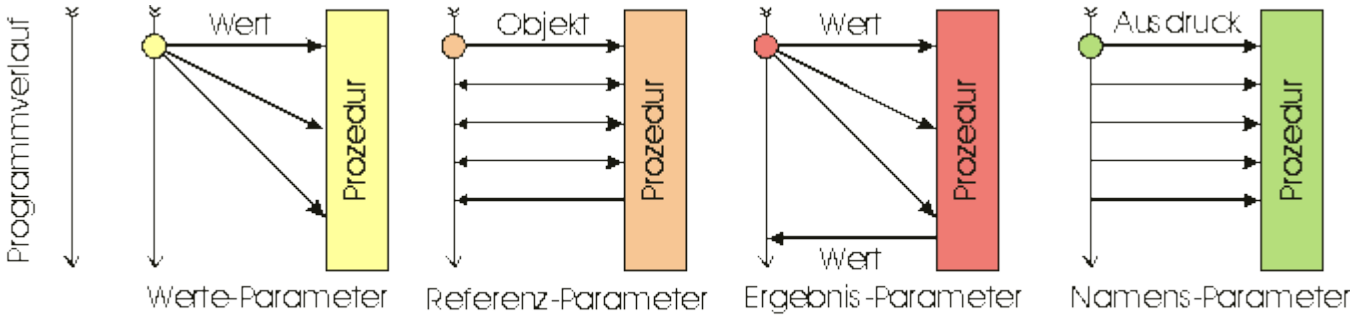
Prozeduren und Funktionen

- Parameterliste
  - Variable Anzahl von Parametern
    - ```
      class VariableArityInJava {
        { out("sum1="+sum(1.0));
          out("sum2="+sum(1.0,2.0));
          out("sum3="+sum(1,3,4,5,6,7,8,9,10));
        }
        double sum(double ... s) {
          double su=0;
          for(int i=0;i<s.length;i++)su+=s[i];
          return su;
      } }
      ```
      ```

Prozeduren und Funktionen

- Parameterliste
  - Übergabe eines Werts / Objekts an Prozedur
    - Werteübergabe
    - ```
      Wert := sin (Winkel + 45);
      ```
    - Referenzübergabe
    - ```
      Swap (Zahl, Wert);
      Zahl, Wert := Wert, Zahl;
      ```
      - **Swap**-Funktion benötigt nicht nur Werte der Objekte, sondern Objekte selbst, um diese verändern zu können.
  - weitere Mechanismen zur Identifikation von Parametern



Prozeduren und Funktionen

- Werteparameter
  - call by value
  - Manche Sprachen (C, Algol68) nur Werteparameter
  - andere durch explizite Adressangabe nachgebildet
  - zunächst die Werte der aktuellen Parameter berechnen
    - Reihenfolge oft nicht festgelegt
    - Seiteneffekte (vermeiden)
    - errechneten Werte der Prozedur übergeben



Prozeduren und Funktionen

- Werteparameter
  - Konstantenparameter
    - Ada: Schlüsselwort in
    - C++: Schlüsselwort const
    - Java: final

```
void Konstante( final double c ) { c = 3;}
```

  - Werteparameter
    - Wert des Parameters wird lokaler Variablen der Prozedur zugewiesen
    - Wert kann innerhalb der Prozedur verändert werden
    - Initialisierungswert
    - meist implizit eingestellt (Ausnahme Algol 60)



Prozeduren und Funktionen

- Referenzparameter
  - *call by reference*
  - Referenz auf Objekt übergeben
  - Objektparameter
  - aktueller Parameter muss Objekt sein
    - keine Konstante oder Ausdruck
    - Falscher Ausdruck

```
swap (Wert*1, Zahl + 0);
```

    - Mehrdeutigkeit der Notation
    - `Zahl := sin (arg); // Wert von arg`
    - `swap(Wert, Zahl);  // Objekt Wert und Zahl`



Prozeduren und Funktionen

- Referenzparameter
  - Referenzparameter binden weiteren Namen an Objekt
  - verschiedene Namen verweisen auf gleiches Objekt
    - undurchsichtige Programme

```
int Nummer := 1, Groesse := 3;
void Clear(reference int Zahl) {
  Zahl := 0;   //Objekt Zahl und Nummer gleich
  Nummer := 1; // „globales“ Objekt Nummer
}
Clear(Nummer);
Clear(Groesse);
```



Prozeduren und Funktionen

- Referenzparameter
  - Adressen als Konstante

Algol68	C

`proc Name(ref int Variable);`
`begin Variable := ... end;`
`...`
	`Name(int * Variable);`
	`{ * Variable = ... };`
	`...`

`Name(AndereVariable);`	`Name(& AndereVariable);`



Prozeduren und Funktionen

- Referenzparameter
  - Adressen als Konstante
  - ```
    swap(int * WertA; int * WertB) {//Vertausche Werte der Objekte
      int Hilf;
      Hilf := * WertA;        //„WertA" ist das Objekt (Adresse)
      * WertA := * WertB;     //* WertA ist der Wert des Objekts.
      * WertB := Hilf;
    }
    void Rufe(int * WertA; int * WertB){
      swap (WertA, WertB);    // WertA , WertB sind Objekte
    }
    int WertA=1, WertB=2;
    main(){
      swap (& WertA, & WertB) // & WertA , & WertB sind Objekts.
    }
    ```


    ```

Prozeduren und Funktionen

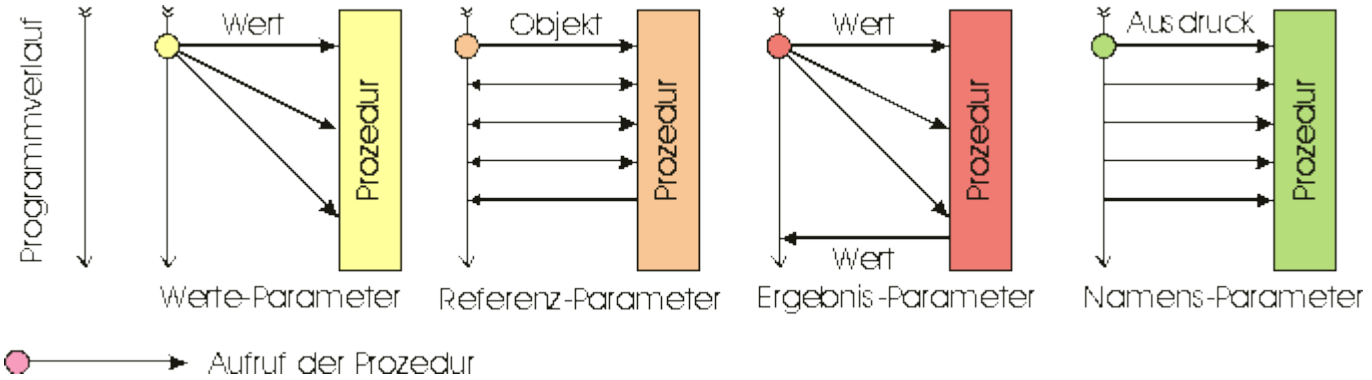
- Referenzparameter
  - Referenzoperator in C++

```
void AddIt(int & x)
{ x += 2; }
int x = 3;
AddIt(x);
printf("%d, x) ; // the value of x is now 5
```



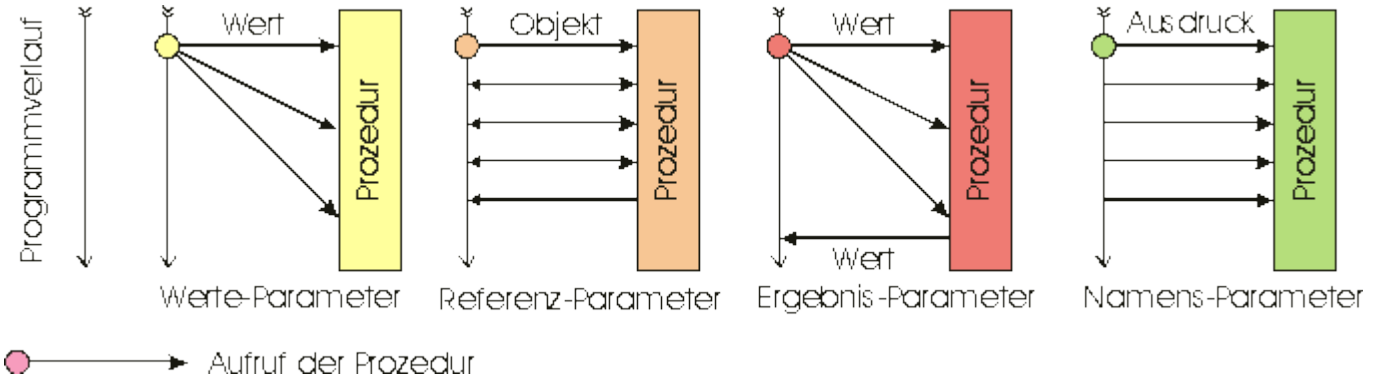
Prozeduren und Funktionen

- Ergebnisparameter
  - Wertergebnisübergabe
    - call by value/result
    - call by copy-in copy-out
    - call by copy-return.
  - zunächst Wert (des Objekts) des aktuellen Parameters berechnet
  - Wert wird lokaler Variablen zugewiesen
  - am Ende der Prozedur wird letzter Wert auf Objekt des aktuellen Parameters übertragen



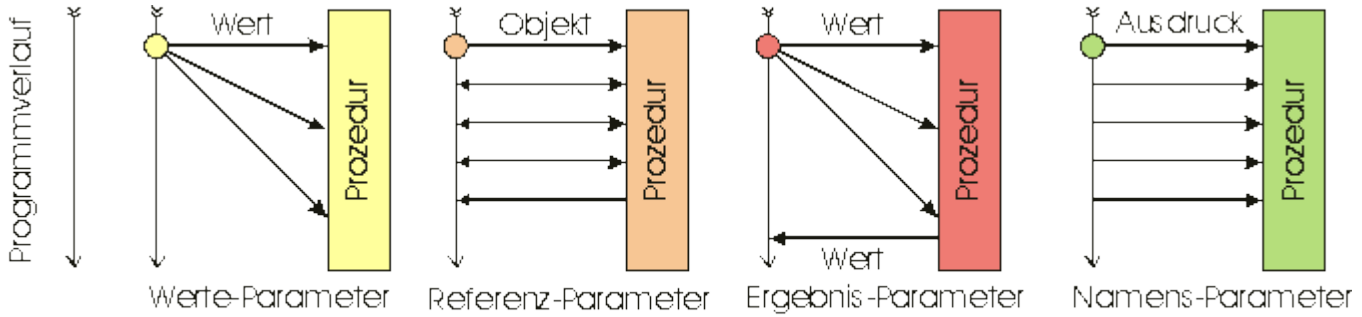
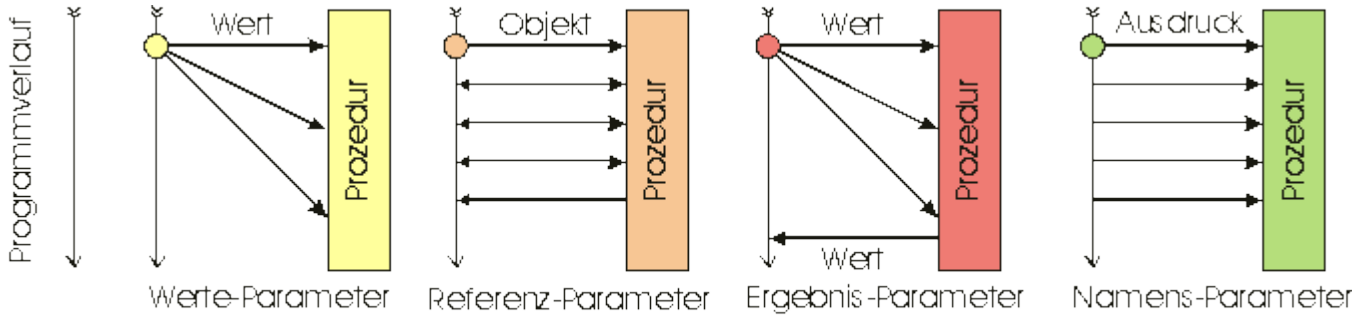
Prozeduren und Funktionen

- Ergebnisparameter
  - Wertergebnisübergabe
    -
```
void Doppel(valueresult integer Zahl1, Zahl2){
   Zahl1 := Zahl1 * 2;
   Zahl2 := Zahl2 * 2;
}
...
{  int Zahl := 2;
   Doppel( Zahl, Zahl);
}
```



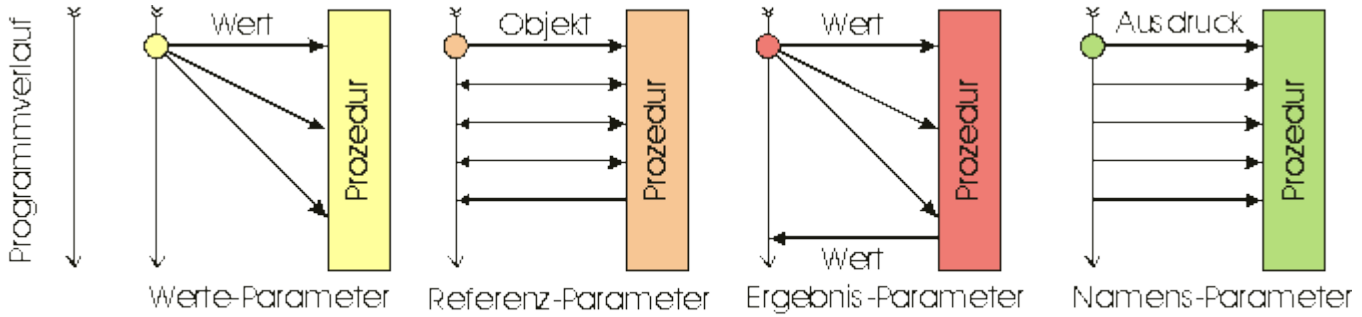
Prozeduren und Funktionen

- Ergebnisparameter
  - Wertergebnisübergabe
    - ```
      void Doppel(valueresult integer Zahl1, Zahl2){
        Zahl1 := Zahl1 * 2;
        Zahl2 := Zahl2 * 2;
      }
      ...
      {  int Zahl := 2;
         Doppel( Zahl, Zahl);
      }
      ```
    - Bei Wertergebnisübergabe: **`Zahl=4`**, bei Referenzparameter: **`Zahl=8`**,


      ```

Prozeduren und Funktionen

- Ergebnisparameter
  - Wertergebnisübergabe
    - ```
      void Doppel(valueresult integer Zahl1, Zahl2){
        Zahl1 := Zahl1 * 2;
        Zahl2 := Zahl2 * 3;
      }
      ...
      {  int Zahl := 2;
         Doppel( Zahl, Zahl);
      }
      ```
    - Reihenfolge des Rückschreibens wichtig!


      ```

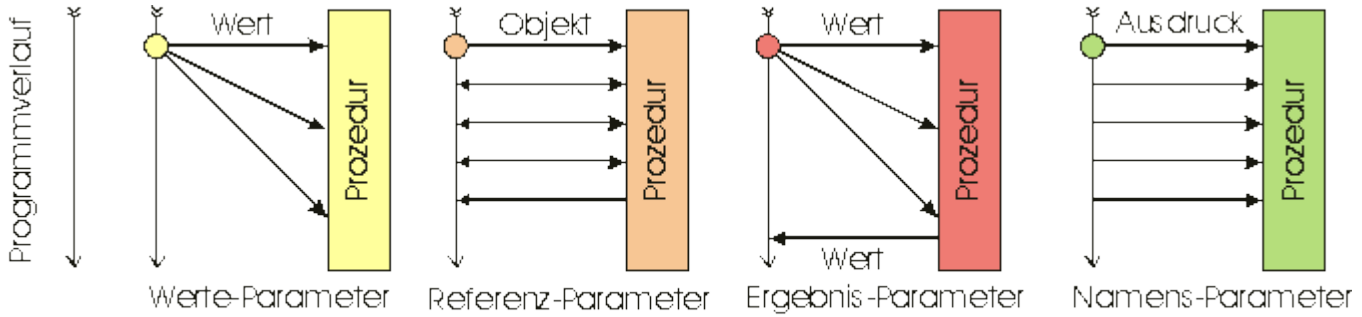
Prozeduren und Funktionen

- Ergebnisparameter
  - Wertergebnisübergabe
    - häufig bei Funktionen auf verschiedenen Rechnern verwendet
    - Referenz auf gleiches Objekte nicht einfach zu erkennen
    - Kommunikationsaufwand eingespart



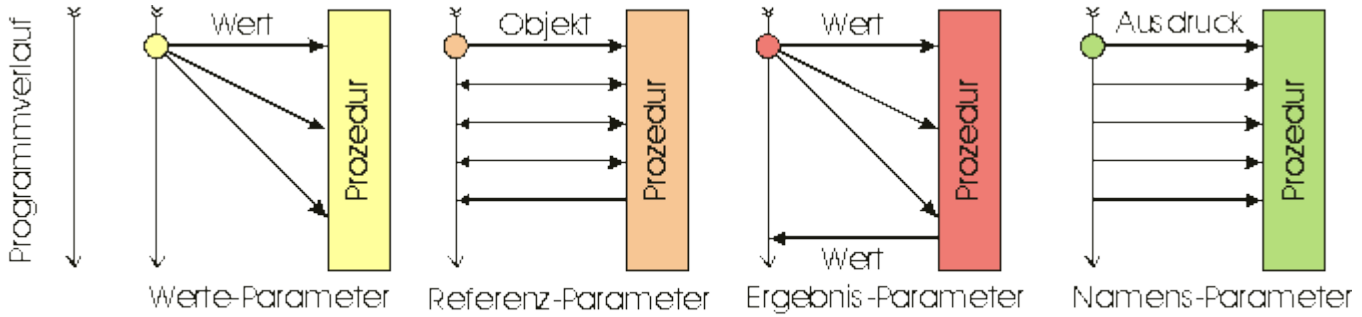
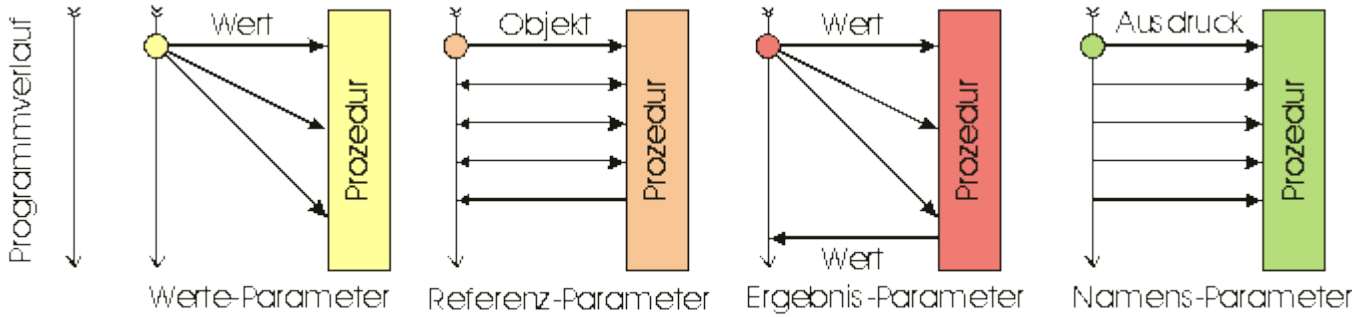
Prozeduren und Funktionen

- Namensparameter
  - in Algol60 eingeführt
    - Aktualwertparameter
  - jüngere Sprachen verwenden sie meist nicht
  - in einigen funktionalen Sprachen wieder eingeführt
    - *delayed evaluation*
  - Analog Referenzparameter, jedoch statt Variable Ausdruck!

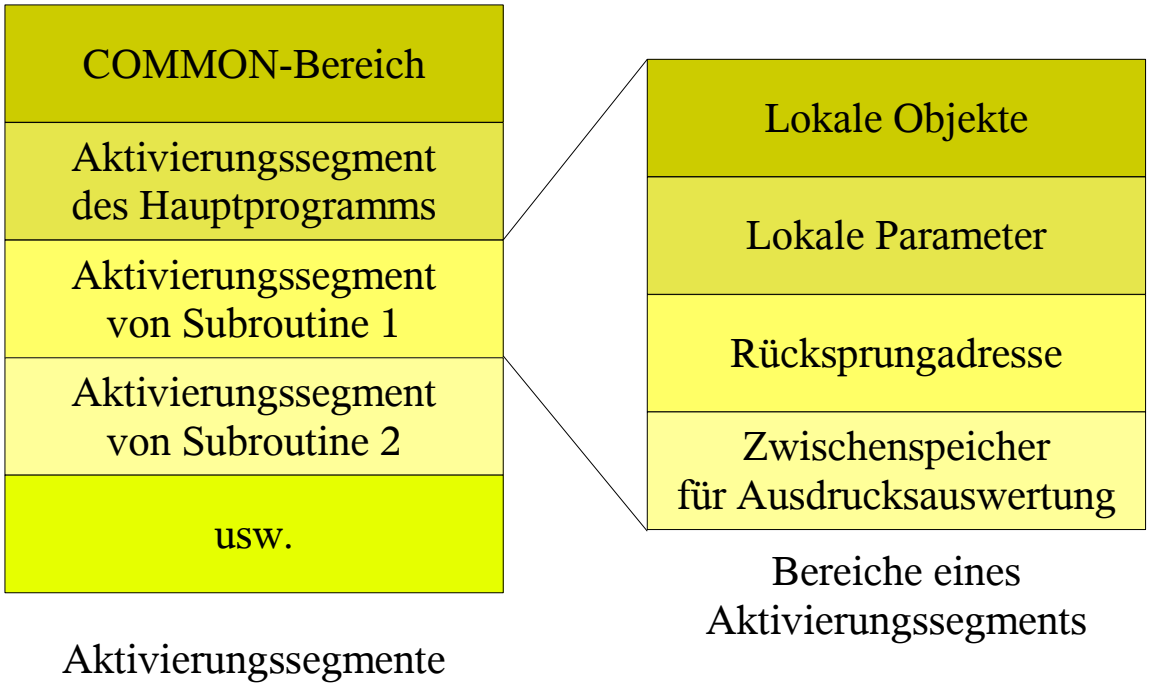


Prozeduren und Funktionen

- Namensparameter
  - ```
    double Summiere(reference double Nr;
                    const integer Von, Bis;
                    name real Wert) {
        double Summe = 0;
        for( Nr = Von; Nr<=Bis; Nr++) Summe += Wert;
        return Summe;
    }
    {
      double Nummer;
      Summe = Summiere(Nummer, 1, 100, Nummer*Nummer);
    }
    ```


    ```

Prozeduren und Funktionen

- Namensparameter
  - sehr flexibel
  - schwierig zu implementieren
    - aktuelle Ausdrücke als eigene Funktionen realisiert
      - in Umgebung des Aufrufs der Funktion
    - Ersetzung des Aufrufs durch Rumpf der Funktion
      - Namensparameter textuell durch aktuellen Ausdruck ersetzen
    - Makro!
      - ursprüngliche Intention?!
    - Inline Function



Prozeduren und Funktionen

- Typen von Parametern
  - Formale und aktuelle Parameter müssen im Typ übereinstimmen
  - In Java: automatische Anpassung
    - ```
      void m(double a){}; // 1
      m(1.0); // ruft 1
      m(2);   // ruft 1
      void m(int a){};    // 2
      m(1.0); // ruft 1
      m(2);   // ruft 2
      ```
    - Probleme bei Neudefinition mit Subtyp.
  - In C
    - Weder Typüberprüfung noch Parameteranzahl
  - In C++
    - Exakte Typüberprüfung, Überladung
      ```

Prozeduren und Funktionen

- Typen von Parametern
  - Referenzübergabe
    - Typgleichheit verlangt
      - Objekte müssen gleichen formalen Typ besitzen
      - Implementierungsprobleme
      - ```
        class typAnpassung {
          { m(1.0); // ruft 1
            m(2);   // ruft 1 // 2
            m(1.0); // ruft 1
            m(2);   // ruft 1 // 2
          }
          void m(double a){out("m(double a) = "+a);} // 1
          //void m(int a) {out("m(int a) = "+a);}    // 2
        }
        ```
        ```

Prozeduren und Funktionen

- Ergebnisübergabe
  - Prozedurparameter liefern Ergebnisse
    - keine Werte in Ausdrücken
      - tangens := sin(x) / cos(x);
    - ohne Funktionen
      - sinx(x, Sinus);
      - cosx(x, Cosinus);
      - tangens := Sinus / Cosinus;
    - Umständlich
    - widerspricht gewohnter mathematischer Notation
      - Strukturierung von Programmen durch Verwendung standardisierter Ausdrücke
    - Programmiersprache sehen Funktionen

Prozeduren und Funktionen

- Funktionskonzept
  - in meisten Programmiersprachen
    - in Fortran
      - Funktion syntaktisch anders definiert als Subroutine
      - nur parametrisierte arithmetische Ausdrücke
  - Funktionen als Erweiterungen von Prozeduren
    - Typen der Ergebnisse von Funktionen meist beschränkt
    - nur skalare Typen wie Ganzzahl oder Gleitpunktzahl
    - komplexe Typen wie Listen über Referenzrückgabe
      - hinderlich
      - einfacher zu implementieren
    - Definition einer Funktion definiert Art von Rückgabewert

Prozeduren und Funktionen

- Funktionskonzept
  - meist nur ein Rückgabewert
    - Listennotation für Variablen
      - Quotient, Rest := GanzzahlDivision(Dividend, Divisor);
      - Wert, Index := MaximumIndex(Feld);
    - Beta [Knudsen93]
      - Birthwistle, Erfinder von Simula67
      - Manipulation ganzer Modelle mit einer Folge von Prozeduren

Prozeduren und Funktionen

- Aufruf von Prozeduren und Funktionen
  - gemeinsam genutzte Anweisungsfolgen
    - Programme strukturierter gestalten
  - komplexe Abläufe durch verbalen Kommentar erklärt
    - besser im Rahmen einer Prozedur strukturieren
  - Prozeduraufruf (procedure call)
    - Schlüsselwort CALL (Fortran)
  - Aufruf durch Prozedurnamen und Parameter
    - seit Algol60
    - Caller
      - aufrufende Anweisung einer Prozedur
    - Callee
      - gerufene Prozedur

Prozeduren und Funktionen

- Aufruf von Prozeduren und Funktionen
  - Funktionen syntaktisch wie Wert nutzbar
    - `Wert := Zufall * 2 + 1;`
  - ist Zufall Variablenname oder Funktionsaufruf?
    - bei jeder Ausführung neuer, zufälliger Wert für Zufall?
  - Funktionsaufruf syntaktisch anders als Referenzierung einer Konstanten oder Variablen
  - C, C++, Java
    - `Wert := Zufall() * 2 + 1;`
  - leere Klammer zeigt, dass es sich um eine Funktion handelt.
  - Klammern mit Parametern
    - `Wert := sin(Winkel)* Laenge;`

Prozeduren und Funktionen

- Aufruf von Prozeduren und Funktionen
  - Funktionen syntaktisch wie Wert nutzbar
    - in Ada
      - Parameter darf wahlweise fehlen
      - bei Prozeduraufrufen ohne Ergebnisrückgabe keine leeren Klammern
      - Prozeduraufrufe treten syntaktisch nicht in Position eines Werts auf
    - möglichst viel Funktionalität in Prozedur?
      - bei gleichem Namen große Anzahl unterschiedlicher Kombination von Parametern
      - Funktionalität zusätzlich durch Konstante gesteuert
        - nicht sehr hilfreich,
          - Funktionalität beim Aufruf schwer zu erkennen.

Prozeduren und Funktionen

- Aufruf von Prozeduren und Funktionen
  - Funktionen syntaktisch wie Wert nutzbar
    - Überladen von Funktionsnamen
      - C++, Ada und Java
      - C in OpenGL
        - ```
          void glColor3f(float r, float g, float b);
          void glColor4f(float r, float g, float b, float a);
          void glColorfv(float r[]);
          ```
      - Aklternativ Überladung
        - ```
          void glColor(float r, float g, float b);
          void glColor(float r, float g, float b, float a);
          void glColor(float r[]);
          ```
      - nicht immer nützlich
      - nur bei ähnliche Funktionalität, ähnliche Operatoren
    - mathematische Programmbibliotheken
      - enthalten viele Prozeduren und Funktionen
      - Funktionalität nach tieferliegenden mathematischen Methoden

Prozeduren und Funktionen

- Umgebung einer Prozedur
  - Ausführungsumgebung
    - *execution environment*
    - Blöcke werden in Umgebung ihrer Spezifikation ausgeführt
  - bei Prozeduren
    - Anweisungsfolge des Prozedurrumpfes steht an anderen Stelle als Programmumgebung des Prozeduraufrufs
    - Sprache legt fest
      - in welcher wird Rumpf einer Prozedur ausgeführt
        - *dynamische Programmumgebung*
          - Umgebung des Aufrufs einer Prozedur
        - *statische Programmumgebung*
          - Umgebung der Definition einer Prozedur
      - Programmiersprachen unterscheiden sich u.a. darin, in welche Programmumgebung eine gerufene Prozedur eingefügt wird.

Prozeduren und Funktionen

- Umgebung einer Prozedur
  - im Prinzip drei Möglichkeiten
    1. Prozedur dynamisch an die Umgebung des Aufrufs gebunden
    2. Prozedur statisch an die definierende Umgebung gebunden,
    3. Prozedur in neutraler Umgebung eingebunden
       - keine Bezug zu einer anderen Umgebung
       - Sicherste Möglichkeit
       - am einfachsten zu implementieren
       - unüblich
         - IMPORT in Modula-2
         - COMMON-Anweisung in Fortran

Programmumgebung	Speicherverwaltung (Laufzeitumgebung)
neutral	statisch
statisch	stackbasiert
dynamisch	allgemein

Prozeduren und Funktionen

- Umgebung einer Prozedur
  - statische Speicherverwaltung
    - Aktivierungssegment (activation record)
      - speichert lokale Objekte
      - nur statische Objekte



Prozeduren und Funktionen

- Umgebung einer Prozedur
  - stackbasierte Speicherverwaltung
    - rekursiv (recursive) Prozeduren
      - Prozedur ruft sich selbst auf
      - lokale Umgebung darf nicht zerstört werden
      - relativ einfache stackbasierte Speicherverwaltung
        - gerufene Prozedur legt sich dynamisch Aktivierungssegment an
        - nicht an fester Stelle im Speicher
        - Aktivierungssegmente im Stack angehängt
        - wird Block/Prozedur verlassen, so wird letztes Aktivierungssegment entfernt
        - das vorherige Aktivierungssegment wird wieder gültig
        - optimale Speicherplatzbelegung
        - alle gültigen Objekte aktuell verfügbar
        - zusätzlicher Aufwand beim Ablauf der Prozeduren

Prozeduren und Funktionen

- Umgebung einer Prozedur
  - stackbasierte Speicherverwaltung
    - bei statischer Bindung
      - Feldes von Zeigern auf statische Umgebung von Blöcken
    - lokale Objekte werden in Aktivierungssegment gehalten
      - Zugriff über Stackzeiger
      - Stackzeiger zeigt auf aktuelles Aktivierungssegment
        - Aktivierungssegment
          - Adresse, an welche die Prozedur zurückkehren soll
          - formale Parameter
          - Zwischenspeicher zur Berechnung von Werten
          - Rückgabewerte bei Funktionen

Prozeduren und Funktionen

- Umgebung einer Prozedur
  - allgemeine Speicherverwaltung
    - hängende Referenzen bei Adressen lokaler Variablen
    - Aktivierungssegment nicht sofort löschen
      - Lisp
        - Garbagecollection
      - Aktivierungssegmente baumartig organisiert
      - im Heap abgelegt
    - Ergebnis Prozedur
      - Modula-2
        - Adresse auf lokale Prozedur
        - lokale statische Umgebung verschwindet, sobald Prozedur beendet
        - deklarierte Variable verschwunden
        - fehlerhafter Zugriff
        - nur globale Prozeduren als Ergebnis einer Funktion erlaubt

Prozeduren und Funktionen

- Dynamische Umgebung einer Prozedur
  - Typ der Umgebungsvariablen kann sich ändern
    - ```
      // a nicht bekannt!
      double f(double x) {return x*a;}
      {  int     a = 3;
         double  b = f(5); // b = 5*3 = 15
      }{ double  a = 7;
         double  b = f(5); // b = 5*7 = 35
      }{ String  a = "ABC";
         double  b = f(5); // b = 5*"ABC" = ?
      }
      ```
    - kann statisch geprüft werden
    - schränkt Verwendbarkeit stark ein
      ```

Prozeduren und Funktionen

- Dynamische Umgebung einer Prozedur
  - Nur möglich wenn
    1. Prozedur verwendet nur lokale Namen
    2. nur ein Namensraum und keine Blockstruktur
    3. keine typisierten Variablen, oder Typ hängt vom Namen ab
  - 1. durch Macros realisierbar
    - in C
      ```
      #define max(A, B) (A > B ? A : B)
      ```
    - Aufruf von
      ```
      Ergebnis := max(Wert, Zahl);
      ```
    - wird expandiert zu
      ```
      Ergebnis := (Wert > Zahl ? Wert : Zahl);
      ```
  - Macros ohne Typüberprüfung
  - meist nicht sehr sicher!

Prozeduren und Funktionen

- Dynamische Umgebung einer Prozedur
  - Nur möglich wenn
    1. Prozedur verwendet nur lokale Namen
    2. nur ein Namensraum und keine Blockstruktur
    3. keine typisierten Variablen, oder Typ hängt vom Namen ab
  - 2. siehe Fortran (allerdings dort nicht realisiert)
  - 3. in einfachen Sprachen wie Basic
    - ```
      // Typ hängt von Bezeichner ab
      name%  // real-Variable
      name#  // int-Variable
      ```
    - bei rekursivem Aufruf
      - Sicher globale Variablen mit gleichem Namen wie lokale Variable auf Stack
    - bei Rücksprung
      - restauriere globale Variablen vom Stack
      ```

Prozeduren und Funktionen

- Dynamische Umgebung einer Prozedur
  - 3. Basic
    - ```
      // Typ hängt von Bezeichner ab
      // sichere lokale Variable a# vor Prozeduraufruf
      int a# = 2;
      double y% = 7.0;
      int b# = f(5.0); // sichert a#=2 auf stack
                       // a# hat jetzt Wert a#=4
                       // restauriert a#=2 von stack
      out << a#;       // gibt 2 aus
      double f(double x%) {
        int a# = 4;
        return x%*a#*y%;  // 5.0*4*7.0
      }
      ```
      ```

Prozeduren und Funktionen

- Dynamische Umgebung einer Prozedur
  - Nachteile von statischer Programmumgebung?
    - jedes lokale Objekt lässt sich über Referenzmechanismus einer Prozedur bekannt machen
      - keine funktionale Einschränkung
    - globale Umgebung lässt sich nicht von jeder dynamischen Laufzeitumgebung aus erreichen
      - Namen könnten verdeckt sein
    - Verwendung der statischen Laufzeitumgebung stellt Erweiterung der Funktionalität dar
    - statische Laufzeitumgebung vorteilhaft gegenüber dynamischer
  - dynamische Laufzeitumgebung
    - meistens schwieriger zu realisieren
    - zusätzliche Laufzeitunterstützung
      - macht Programme langsamer