Technische Universität Braunschweig

Institute of Operating Systems and Computer Networks
Reliable System Software

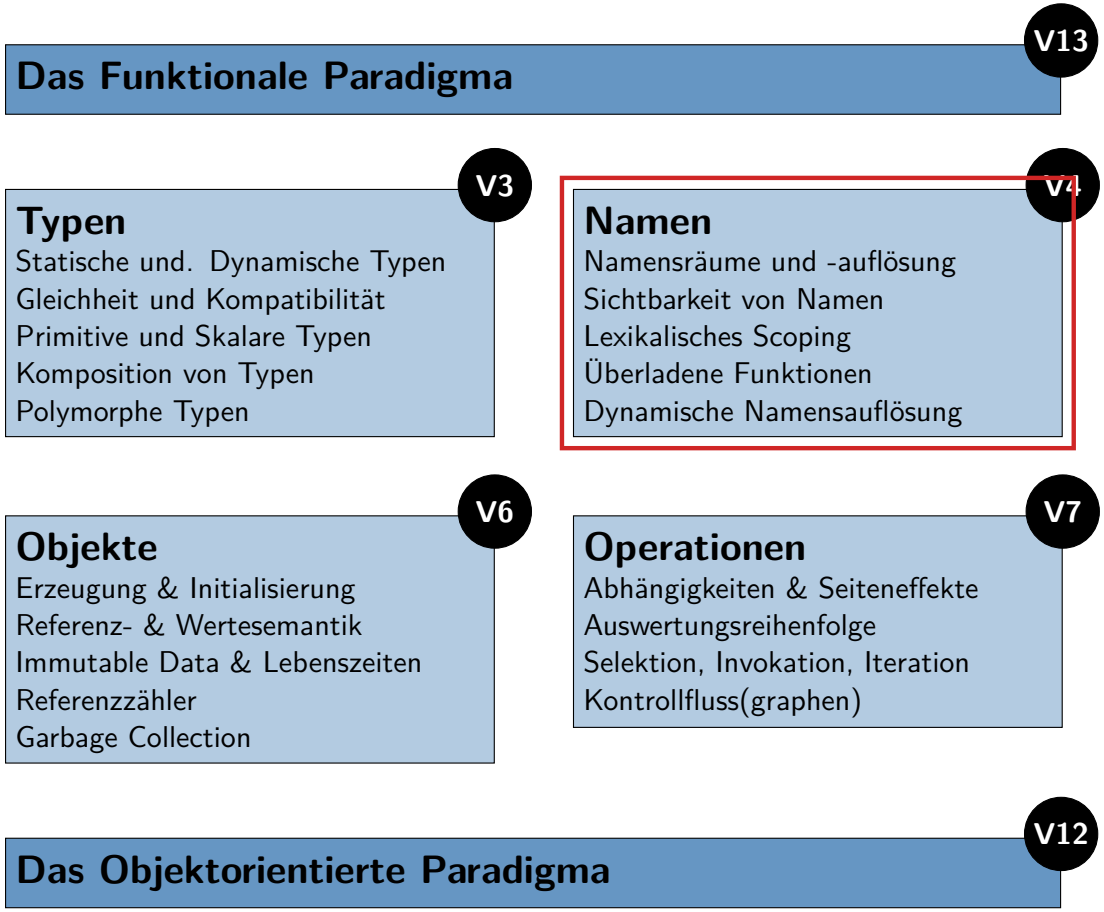
Programmiersprachen und Übersetzer

04 - Namen

Christian Dietrich

Sommersemester 2025

Wieso sollte ich mich mit Namen beschäftigen?



- Die Namensauflösung ist grundlegend für das Programmverständnis.
  - Namen etablieren kontextsensitive Querverbindungen zu Deklarationen.
  - Das informatische Grundprinzip von Information Hiding hängt an Namen.

”Motivierendes” Beispiel: C++

```C++
namespace A {
  typedef int type_t;
  class B {
    private:   int x;
    protected: int y;
    public:    int z;
      virtual type_t func();
  };
}

class C : public A::B {
  virtual A::type_t func();
};

using ret_t = A::type_t;
```

```C++
ret_t A::B::func() {
  return this->x;
}

ret_t C::func() {
  return y;
}

int main() {
  A::B *obj = new C();
  obj->z = 42;
  return obj->func();
}
```

- Namen werden überall im Programm deklariert und referenziert.
- Die **Namensauflösung** verbindet den Namen mit dem Objekt.
- Information sfluss entlang dieser Kanten quer zum AST

”Motivierendes” Beispiel: C++

```
namespace A {
  typedef int type_t;
  class B {
    private:   int x;
    protected: int y;
    public:    int z;
      virtual type_t func();
  };
}

class C : public A::B {
  virtual A::type_t func();
};

using ret_t = A::type_t;
```

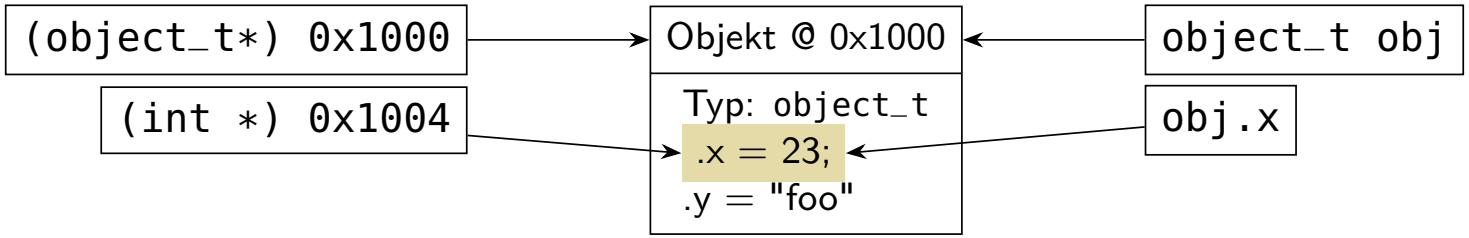
```
ret_t A::B::func() {
  return this->x;
}

ret_t C::func() {
  return y;
}

int main() {
  A::B *obj = new C();
  obj->z = 42;
  return obj->func();
}
```

C++

type = int
addr = this + 4;

- Namen werden überall im Programm deklariert und referenziert.
- Die **Namensauflösung** verbindet den Namen mit dem Objekt.
- Information sfluss entlang dieser Kanten quer zum AST

Grundbegriffe: Namen und Namensauflösung

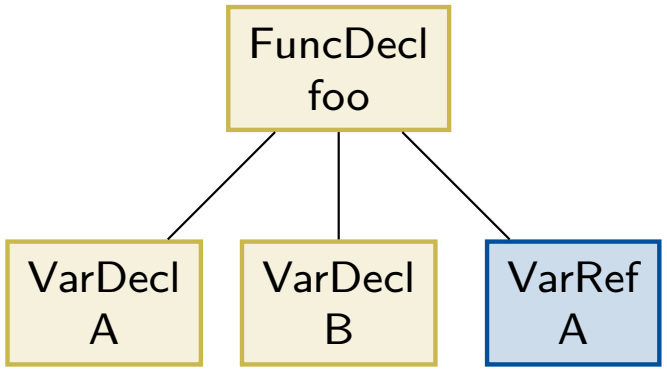


- **Name**: symbolische und statische Referenz im Programmcode
  - Abstraktion: Namen ersetzen die Verwendung konkreter Speicheradressen.
  - Dokumentation: Namen transportieren Informationen zwischen Entwicklern.
  - Querverbindung: Referenz von Deklarationen quer zum AST
- **Namensauflösung**: Auffinden des referenzierten Objekts
  - Wird vom Übersetzer teilweise vorbereitet oder schon ganz durchgeführt
  - Essenzieller Teil der semantischen Analyse (`name 'x' is not defined`)
  - Verwendung statischer (Bezeichner) wie dynamischer (Pointer) Informationen

Im Übersetzer suchen wir (oft) nur die **Deklarationsstelle** des Namens

Grundbegriffe: Deklaration, Definition, Namensraum

```
void foo() {
  int A;
  extern int B;
  A
}
```



- Deklaration eines Namens für einen AST Knoten
  - Um AST Knoten wiederzufinden, geben wir ihnen einen Namen.
  - Viele Konstrukte deklarieren einen Namen (Typen, Variablen, Funktionen).
- Definition von benannten Objekten
  - Jede Definition ist auch eine Deklaration und führt einen Namen ein.
  - Geht einher mit der Allokation eines Objekts (z.B. Definition einer Variablen)
  - Wichtige Unterscheidung bei separater Übersetzung (C/C++)
- Namensraum ist ein Container für deklarierte Namen
  - Beschränken die Sichtbarkeit eines deklarierten Namens
  - Schachtelung: Ein Namensraum kann auch weitere Namensräume beinhalten.

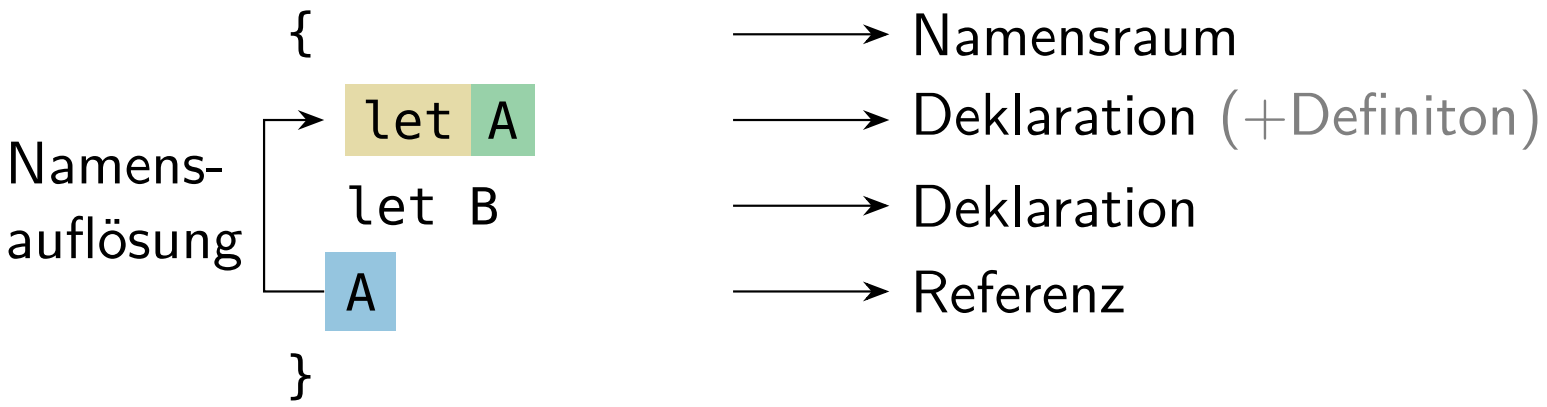
Grundbegriffe: Deklaration, Definition, Namensraum

```
{
  let A
  let B
  A
}
```

Wir werden eine Mini-Sprache entwicklen, um wichtige Aspekte der Namensauflösung zu beleuchten.

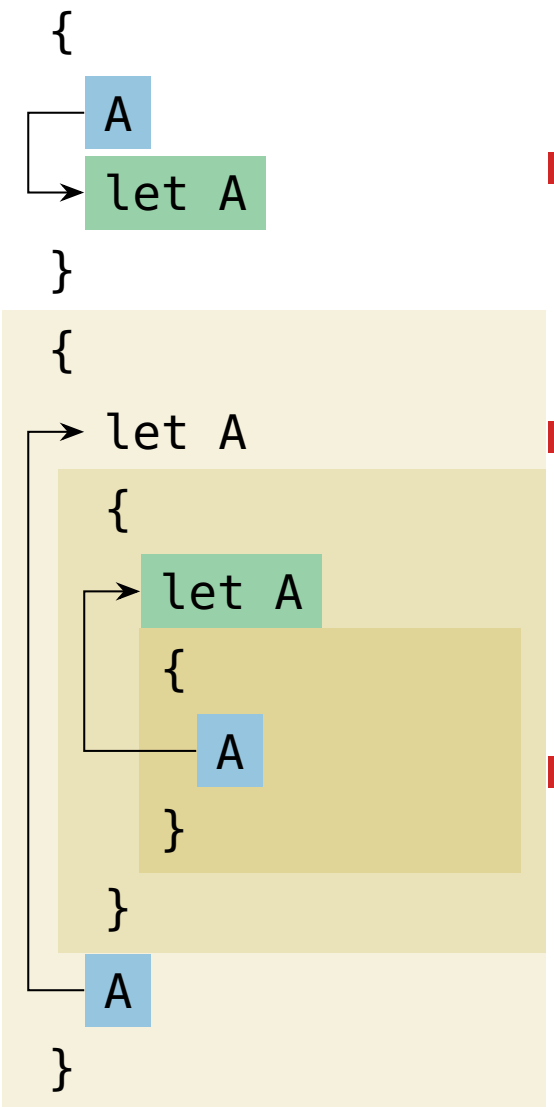
- Deklaration eines Namens für einen AST Knoten `let <NAME>`
  - Um AST Knoten wiederzufinden, geben wir ihnen einen Namen.
  - Viele Konstrukte deklarieren einen Namen (Typen, Variablen, Funktionen).
- Definition von benannten Objekten
  - Jede Definition ist auch eine Deklaration und führt einen Namen ein.
  - Geht einher mit der Allokation eines Objekts (z.B. Definition einer Variablen)
  - Wichtige Unterscheidung bei separater Übersetzung (C/C++)
- Namensraum ist ein Container für deklarierte Namen `{...}`
  - Beschränken die Sichtbarkeit eines deklarierten Namens
  - Schachtelung: Ein Namensraum kann auch weitere Namensräume beinhalten.

Grundbegriffe: Deklaration, Definition, Namensraum



- Deklaration eines Namens für einen AST Knoten `let <NAME>`
  - Um AST Knoten wiederzufinden, geben wir ihnen einen Namen.
  - Viele Konstrukte deklarieren einen Namen (Typen, Variablen, Funktionen).
- Definition von benannten Objekten
  - Jede Definition ist auch eine Deklaration und führt einen Namen ein.
  - Geht einher mit der Allokation eines Objekts (z.B. Definition einer Variablen)
  - Wichtige Unterscheidung bei separater Übersetzung (C/C++)
- Namensraum ist ein Container für deklarierte Namen `{...}`
  - Beschränken die Sichtbarkeit eines deklarierten Namens
  - Schachtelung: Ein Namensraum kann auch weitere Namensräume beinhalten.

Implizite Namensauflösung

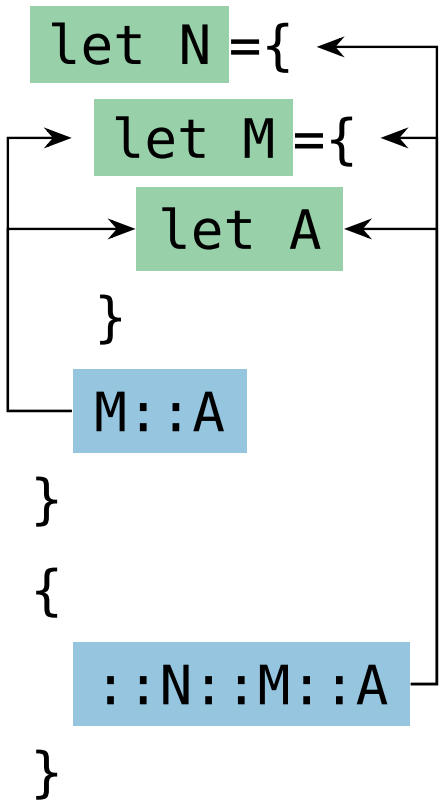


- „Declaration before Use“
  - Deklaration in einem umgebenden Namensraum
  - Reihenfolge kann (teilweise) egal sein
- „Closest-nested scope rule“ (CNSR)
  - Suche nach Deklarationen von innen nach außen
  - Bei mehreren Möglichkeiten: Nimm die innerste
  - Im AST: Wir suchen von unten nach oben.
- Deklarationen können verdeckt werden.
  - Innere Deklarationen verdecken die äußeren.
  - Die äußere Deklaration bleibt bestehen, ist aber innen nicht mehr sichtbar.

Explizite Namensräume

Problem: Referenzen nur entlang des ASTs möglich

Die implizite Namensauflösung erlaubt uns nur Namen zu referenzieren, die direkt über uns im AST sind. Namen aus einem anderen Namensraum (NS) sind nicht erreichbar.



- NS bekommt Namen im umgebenden NS
  - Namen aus einem NS anfordern mit „`::`“
  - Der Trenner `::` ist wie das „`/`“ bei Pfaden
  - Grundlegende Erweiterung des Namensraumkonzepts
- Impliziter Wurzelnamensraum
  - Ermöglicht es, alle Namensräume zu adressieren
  - Vollständig qualifizierter Name ist eindeutig
- In C++: „`let <X>=...`“ = „`namespace <X>`“

```C++
namespace N { namespace M {
    int A;
} }
```

Import von entfernten Namen

Problem: Explizite Auflösungspfade sind zu lange

Ständig den vollständigen Namen (`::A::B::C`) einer entfernten Deklaration zu verwenden, ist umständlich viel Schreibarbeit.

```
let N={  // N_d=A,M
  let A
  let M={  // N_d=B,C
    let B
    let C
  }
}
let X={
  import ::N::A
  import ::N::M::*
  A
  B
}
```

- Jeder NS hat verschiedene Namensmengen
  - Die direkt deklarierten Namen (N_d)
  - Die implizit sichtbaren Namen (N_v)
  - Den Wurzelnamensraum
- Viele Sprachen erlauben Import von Namen
  - `import <PFAD>` erweitert die Namensmengen
  - Explizite Namensliste oder Importe mit `*`
  - Importe geschehen oft implizit (z.B. Vererbung)
- **Bei uns:** Importe nach N_d sind transitiv
  - Importiere Namen können re-importiert werden
  - `import ::X::*` enthält auch `::N::M::B`

Einschränkung der Sichtbarkeit

Ziel: Wir wollen Interna der Implementierung verstecken

Zum Zwecke des **Information Hidings** wollen wir manche Namen nur innerhalb eines Namensraumes sichtbar machen.

- Deklarationen bekommen ein weiteres Attribut `let[I] <NAME>`
  - Interne Deklarationen können nur über implizite Auflösung gefunden werden.
  - Name wird unsichtbar für alles außerhalb des aktuellen NS
  - Wir können interne Namen ändern, ohne externe Komponenten anzufassen.

```
{
  let M={
    let[I] A
    A
  }
  M::A
}
```

- Zugriff auf einen internen Namen ist nicht erlaubt

  Übersetzer findet zwar die Deklaration, meldet aber aufgrund des Attributs einen Fehler.
- Interne Namen werden beim Import übersprungen.

Einschränkung der Sichtbarkeit

Ziel: Wir wollen Interna der Implementierung verstecken

Zum Zwecke des **Information Hidings** wollen wir manche Namen nur innerhalb eines Namensraumes sichtbar machen.

- Deklarationen bekommen ein weiteres Attribut `let[I] <NAME>`
  - Interne Deklarationen können nur über implizite Auflösung gefunden werden.
  - Name wird unsichtbar für alles außerhalb des aktuellen NS
  - Wir können interne Namen ändern, ohne externe Komponenten anzufassen.

```
{
  let M={[I]
    let A
    A
  }
  M::A
}
```

- Zugriff auf einen internen Namen ist nicht erlaubt

  Übersetzer findet zwar die Deklaration, meldet aber aufgrund des Attributs einen Fehler.
- Interne Namen werden beim Import übersprungen.
- Namensräume können Standardattribute haben
  - `let[I]` macht einen Namen nur intern sichtbar.
  - `let[E]` macht einen Namen auch extern sichtbar.

Fallbeispiel: Java Packages

```
package N;
public class A {
  public static
    void f1() { }
}
```

```
package M;
import N.A;

class B {
  static void f2() {
    A.f1();
  }
}
```

```
let N={[I]
  let[E] A={[I]
    let[E] f1
  }
}

let M={[I]
  import ::N::A
  let B={[I]
    let f2={[I]
      A::f1
    }
  }
}
```

Pakete und Klassen sind per default `private`

`public` macht Namen dennoch sichtbar

Java verwendet "."

Lokale Variablen sind nicht von außen sichtbar

Zeitpunkt der Namensbindung

Bisher: Zu einem Namen haben wir bisher nur die Deklaration gefunden, nicht aber das Objekt bzw. seine Adresse.

Aber: Nicht zu jedem Zeitpunkt ist mit einem Namen wirklich ein existierendes Objekt verbunden.

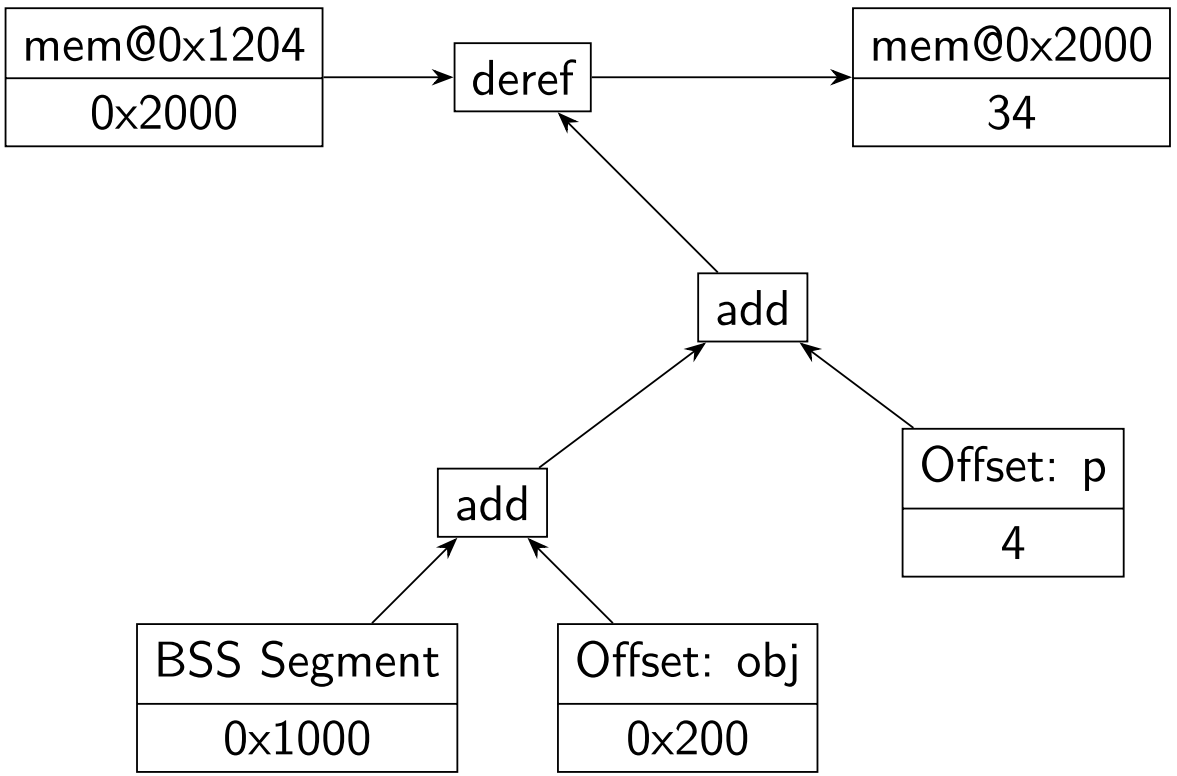
Binding time: Zeitpunkt der Bindung zwischen Name und Objekt

- Compile time: Der Übersetzer kennt Adresse des Objekts bereits.
  Beispiel: Speicherbereich einer Hardwarekomponente.
- Link time: Die Adresse wird beim Erstellen der Binärdatei ermittelt.
  Beispiel: Funktionen und globale Variablen
- Load time: Die Adresse wird beim Starten des Prozesses festgelegt.
  Beispiel: Code in Bibliotheken
- Run time: Die Adresse wird erst zur Laufzeit ermittelt.
  Beispiel: Lokale Variablen und Objekte am Heap

Zeitpunkt der Adressberechnung

- Adressberechnung ist **immer** erst nach der Binding Time möglich.
  - Wenn kein Objekt assoziiert ist, können wir auch keine Adresse ausrechnen.
  - Partielle Namensauflösung: Vorschrift und Werte vorberechnen.

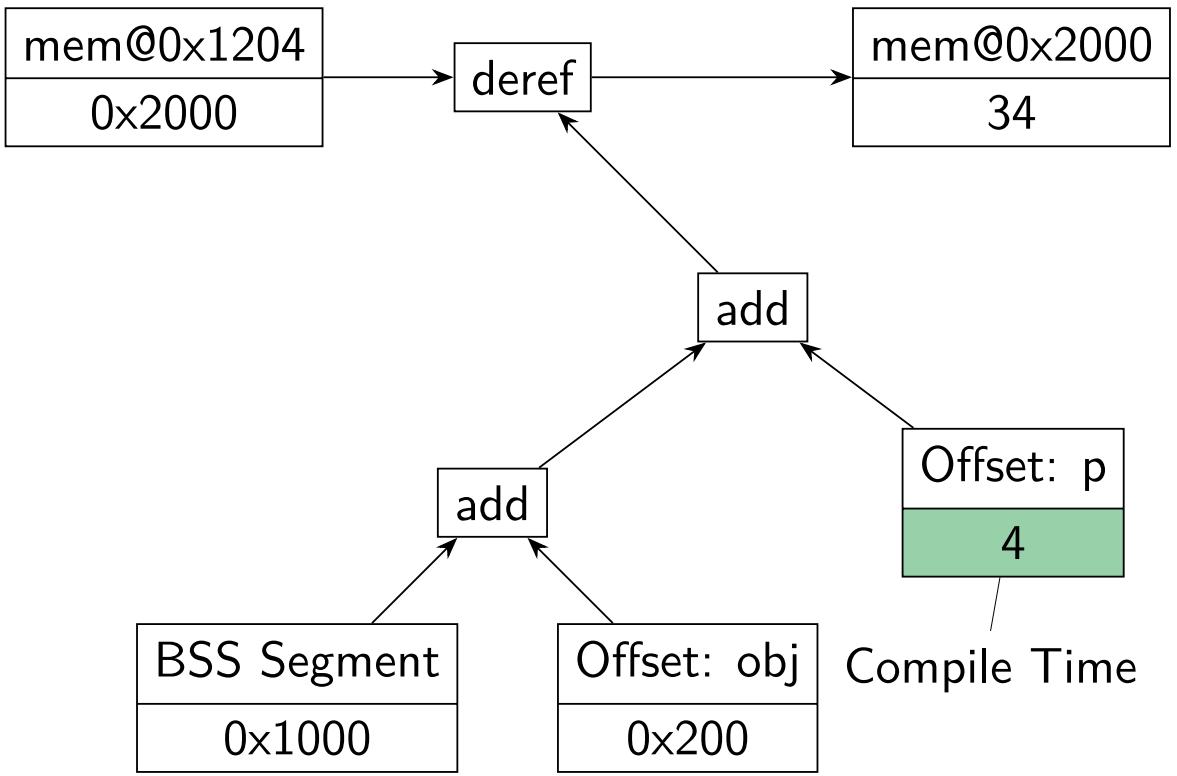
```
struct foo {
  int  a;
  int *p;
} obj;
...
*(obj.p) = ...
```



Zeitpunkt der Adressberechnung

- Adressberechnung ist **immer** erst nach der Binding Time möglich.
  - Wenn kein Objekt assoziiert ist, können wir auch keine Adresse ausrechnen.
  - Partielle Namensauflösung: Vorschrift und Werte vorberechnen.

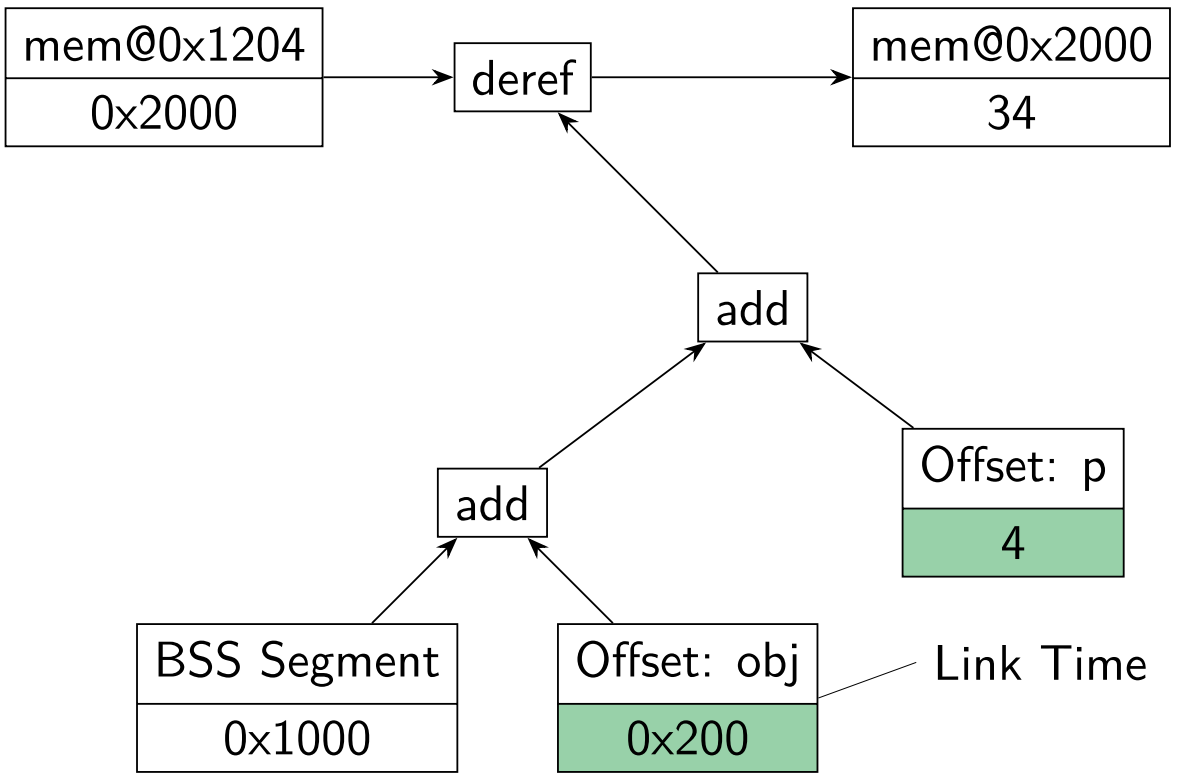
```
struct foo {
  int  a;
  int *p;
} obj;
...
*(obj.p) = ...
```



Zeitpunkt der Adressberechnung

- Adressberechnung ist **immer** erst nach der Binding Time möglich.
  - Wenn kein Objekt assoziiert ist, können wir auch keine Adresse ausrechnen.
  - Partielle Namensauflösung: Vorschrift und Werte vorberechnen.

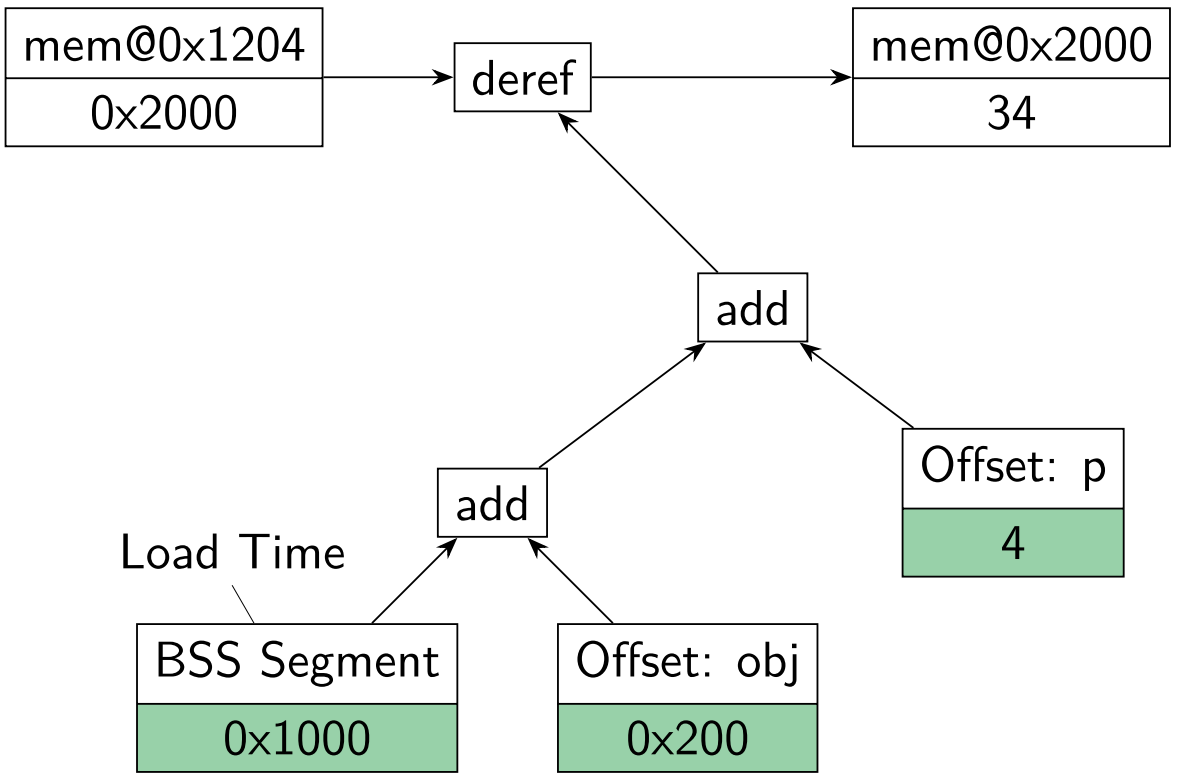
```
struct foo {
  int  a;
  int *p;
} obj;
...
*(obj.p) = ...
```



Zeitpunkt der Adressberechnung

- Adressberechnung ist **immer** erst nach der Binding Time möglich.
  - Wenn kein Objekt assoziiert ist, können wir auch keine Adresse ausrechnen.
  - Partielle Namensauflösung: Vorschrift und Werte vorberechnen.

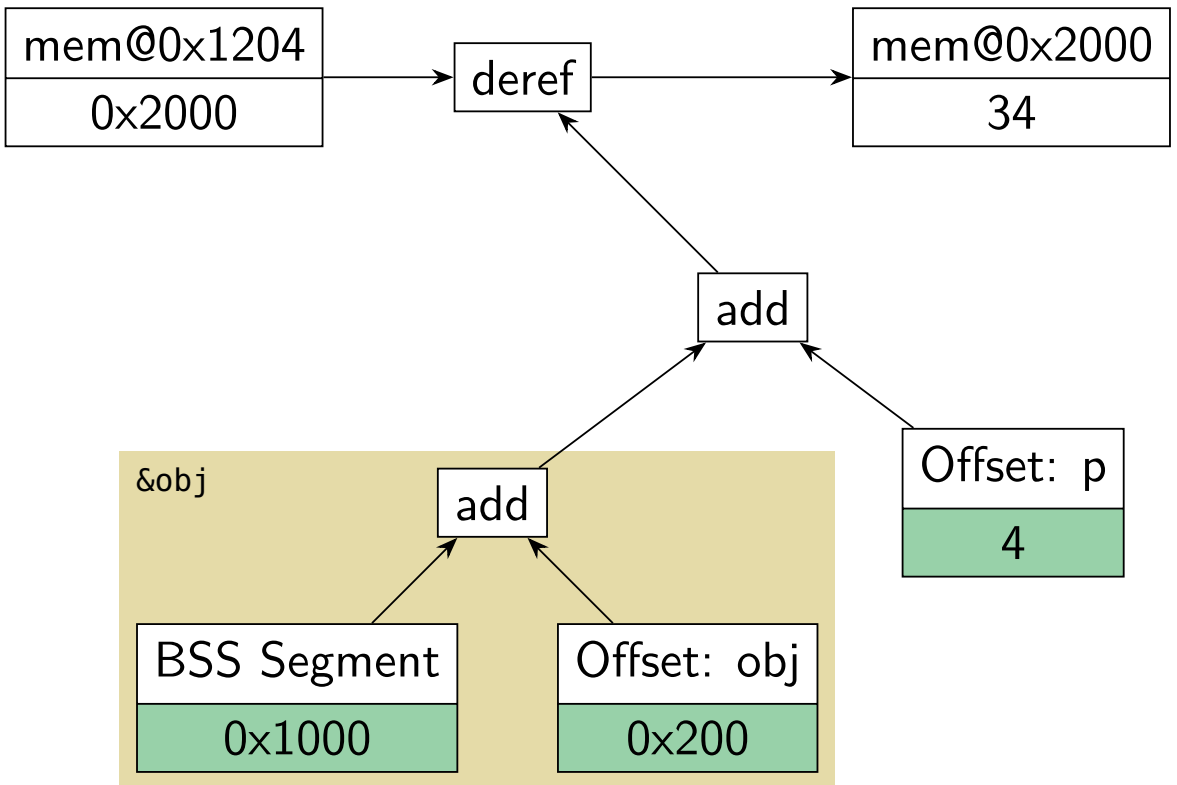
```
struct foo {
  int  a;
  int *p;
} obj;
...
*(obj.p) = ...
```



Zeitpunkt der Adressberechnung

- Adressberechnung ist **immer** erst nach der Binding Time möglich.
  - Wenn kein Objekt assoziiert ist, können wir auch keine Adresse ausrechnen.
  - Partielle Namensauflösung: Vorschrift und Werte vorberechnen.

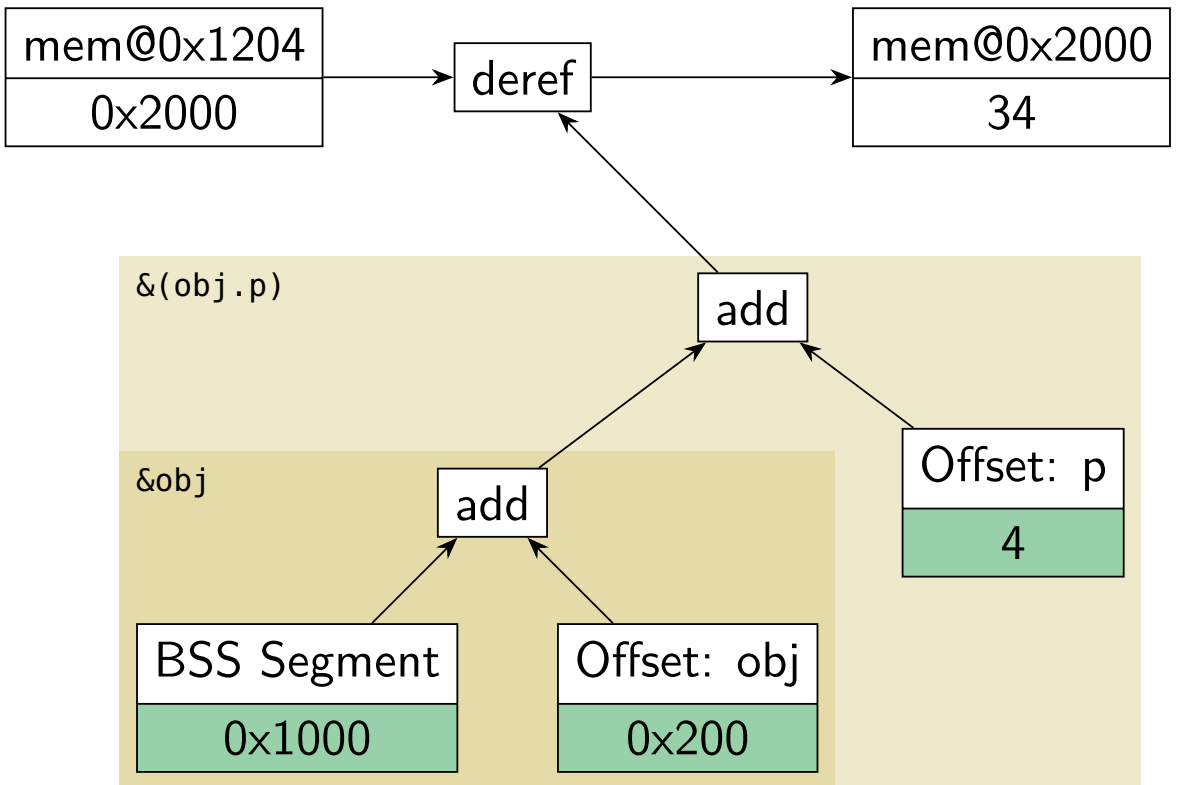
```
struct foo {
  int  a;
  int *p;
} obj;
...
*(obj.p) = ...
```



Zeitpunkt der Adressberechnung

- Adressberechnung ist **immer** erst nach der Binding Time möglich.
  - Wenn kein Objekt assoziiert ist, können wir auch keine Adresse ausrechnen.
  - Partielle Namensauflösung: Vorschrift und Werte vorberechnen.

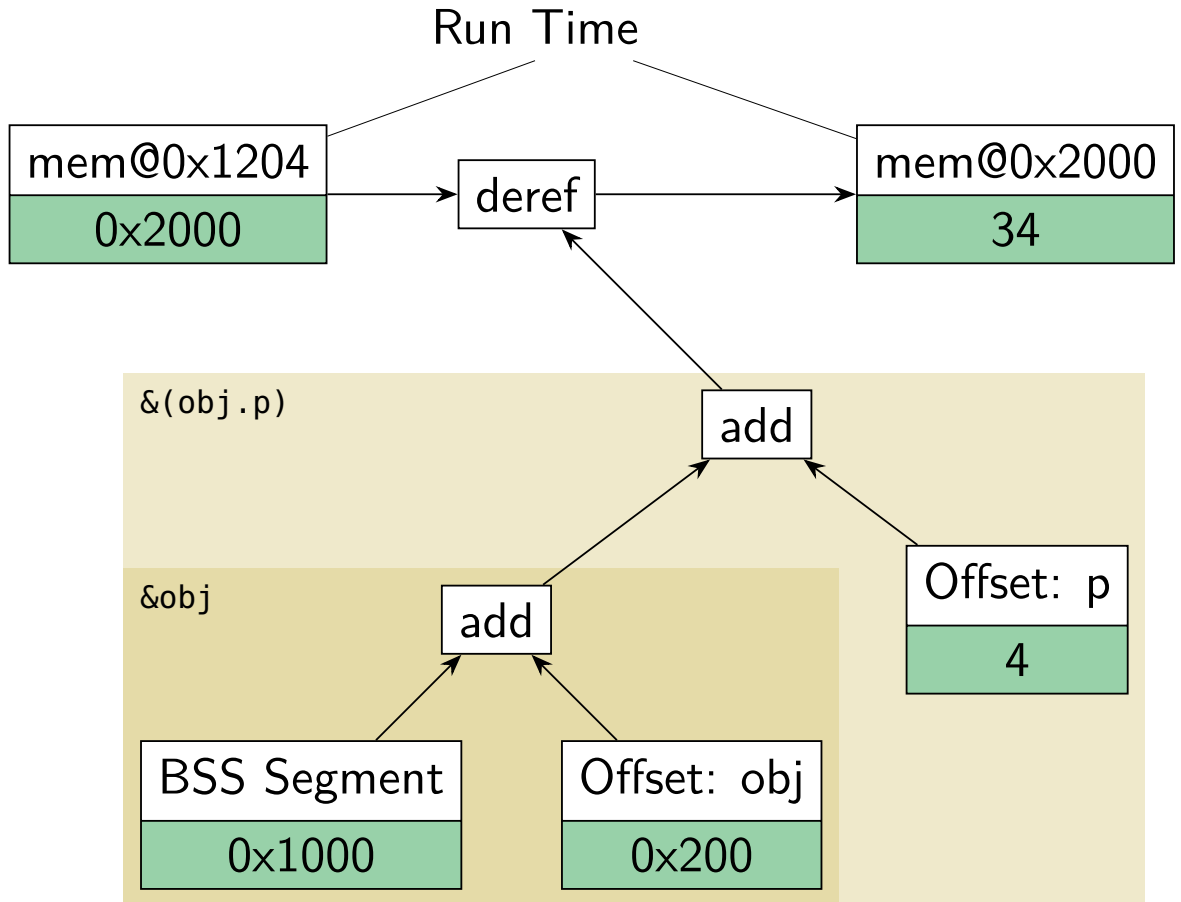
```
struct foo {
  int  a;
  int *p;
} obj;
...
*(obj.p) = ...
```



Zeitpunkt der Adressberechnung

- Adressberechnung ist **immer** erst nach der Binding Time möglich.
  - Wenn kein Objekt assoziiert ist, können wir auch keine Adresse ausrechnen.
  - Partielle Namensauflösung: Vorschrift und Werte vorberechnen.

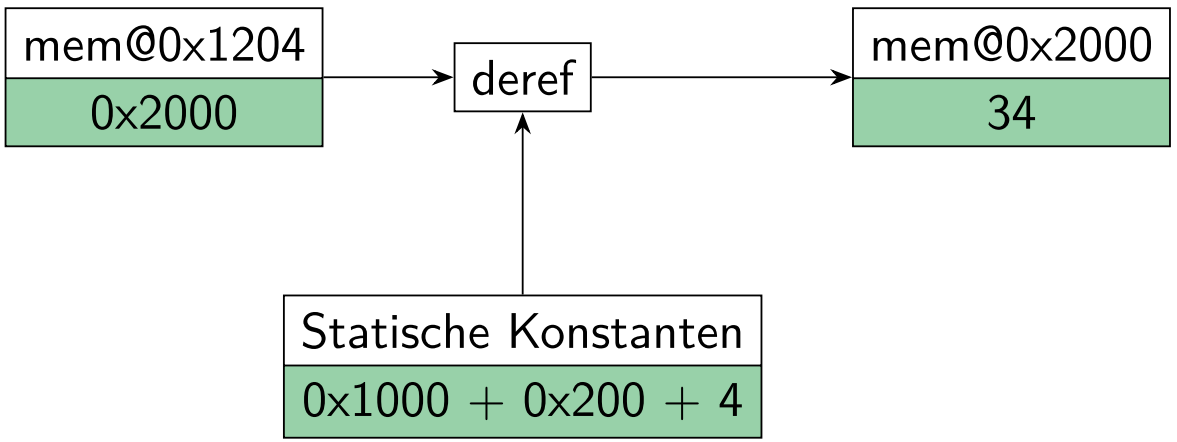
```
struct foo {
  int  a;
  int *p;
} obj;
...
*(obj.p) = ...
```



Zeitpunkt der Adressberechnung

- Adressberechnung ist **immer** erst nach der Binding Time möglich.
  - Wenn kein Objekt assoziiert ist, können wir auch keine Adresse ausrechnen.
  - Partielle Namensauflösung: Vorschrift und Werte vorberechnen.

```
struct foo {
  int  a;
  int *p;
} obj;
...
*(obj.p) = ...
```



Vereinfachung: Wir nehmen im folgenden an, dass es nur Run Time (dynamisch) und Compile Time (statisch) gibt.

Instanziierbare Namensräume

```cpp
namespace N {
  int a;
  int b;
};
N::a
```
C++

Bisher gab es von jedem Namensraum genau eine Instanz. Daher kann jeder enthaltene Name nur ein Objekt referenzieren.

- Namensräume mehrfach als eigene Objekte instanziieren
  - Jedes Laufzeitobjekt enthält eine Bindung für die deklarierten Namen
  - Laufzeitdaten werden zur Unterscheidung gebraucht (Objektadresse)

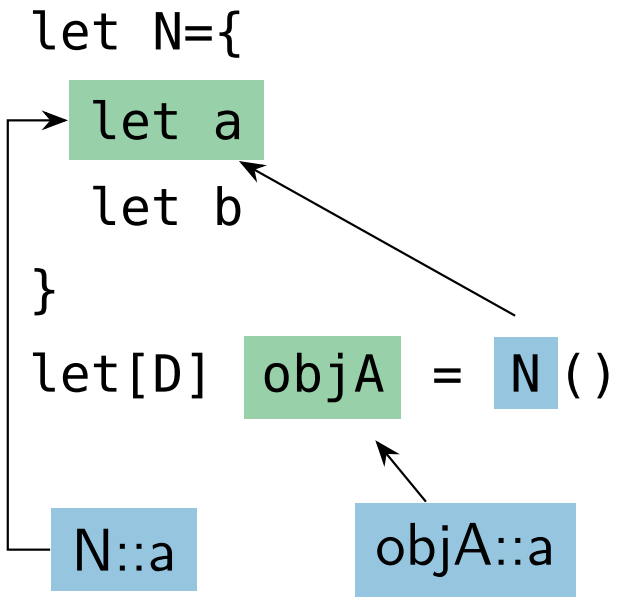
```c
struct N {
  int a;
  int b;
};

struct N objA;
struct N objB;
```
C

- `objA.a` und `objB.a` sind verschieden
- Punktoperator macht dynamische Auflösung `<obj>.<name>`: `&obj + offset(N::name)`
- Java unterscheidet syntaktisch nicht zwischen dynamischer und statischer Namensauflösung.

- Instanziierbare Namensräume sind ein Schlüsselkonzept der Informatik

Instanziierbare Namensräume (II)

```
struct N {
  int a;
  int b;
};

struct N objA;
```

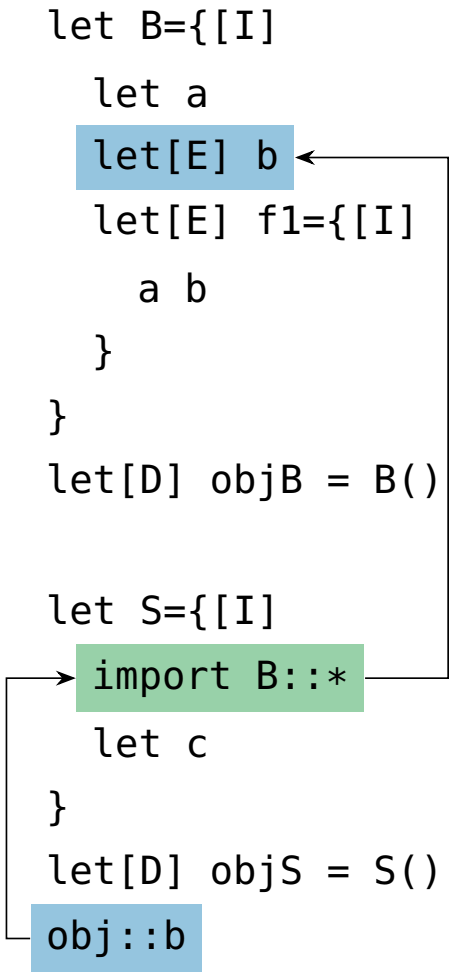


- In unserer minimalen Namensraumsprache führen wir let[D] ein.
  - `let[D] <DST> = <SRC>` kopiert den NS `SRC` unter dem Namen `DST`
  - Wir verwenden weiterhin nur den `::` Operator.
  - Die Auflösung in einem `let[D]` Namensraum erfordert **d**ynamisches Wissen.

Fallbeispiel: Klassen in C++

```
class B {
  int a;
 public:
  int b;
  int f1(){
    return a+b;
  }
};
B objB;
```

```
class S : public B {
  int c;
}
S objS;
```



Kombination aus:

- Namensräumen
- Sichtbarkeit
- Instanziierung
- Import

Feinheiten:

- `protected`-Deklarationen ändern ihre Sichtbarkeit beim vererben.
- `struct` ist per default `[E]`.
- Man kann `public`, `protected` und `private` erben.

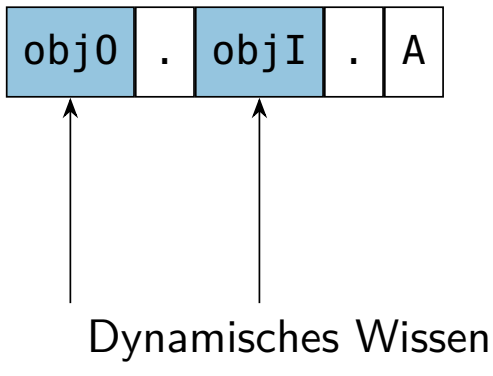
Statische Felder in Klassen

Beichte des Dozenten: Ich habe `static` lange nicht verstanden.

- In Klassenobjekten wird jeder Name dynamisch aufgelöst (this-Zeiger).
  - Manchmal sollen Namen dennoch dasselbe Objekt referenzieren.
  - `let[S]` setzt die dynamische Namensauflösung außer Kraft.

```
class O { public:
 class I { public:
   int A;
 } objI;
} objO;                    C++
```

```
let O={[I,D]
  let I={[I,D]
    let[E] A
  }
}
```



- Statische lokale Variablen in C/C++ funktionieren genauso.
  - Die dynamische Information um den Call-Frame wird ignoriert.
  - Die Variable behält, weil sie immer das selbe Objekt referenziert, ihren Wert.

```
int counter() { static int i = 0; return i++; }
```

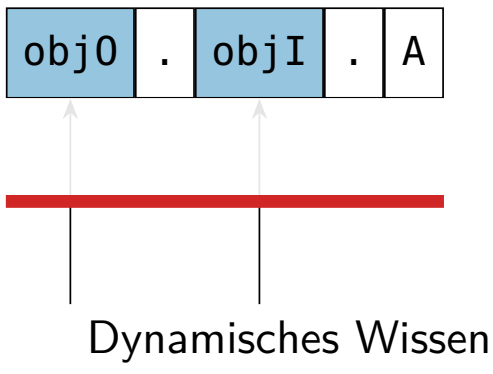
Statische Felder in Klassen

Beichte des Dozenten: Ich habe `static` lange nicht verstanden.

- In Klassenobjekten wird jeder Name dynamisch aufgelöst (this-Zeiger).
  - Manchmal sollen Namen dennoch dasselbe Objekt referenzieren.
  - `let[S]` setzt die dynamische Namensauflösung außer Kraft.

```C++
class O { public:
 class I { public:
  static int A;
 } objI;
} objO;
```

```
let O={[I,D]
  let I={[I,D]
    let[E,S] A
  }
}
```



- Statische lokale Variablen in C/C++ funktionieren genauso.
  - Die dynamische Information um den Call-Frame wird ignoriert.
  - Die Variable behält, weil sie immer das selbe Objekt referenziert, ihren Wert.

```C++
int counter() { static int i = 0; return i++; }
```

Funktionen und lokale Variablen

```c
int f1(int L) {
  return L+1;
}
```
C

```
let f1={[I]
  let L; L
}
```

- Funktionen öffnen für ihren Rumpf einen eigenen Namensraum.
  - Lokale Variablen (und Parameter) sind nur innerhalb der Funktion sichtbar.
  - Die Namen lokaler Variablen können globale verdecken. (Folge von CNSR)
  - Parameterdeklarationen werden zu Namen innerhalb der Funktion.

```c
int fak(int n) {
  if (n < 1) {
    return 1;
  }
  int tmp = fak(n-1);

  return n * tmp;
}
```
C

Rekursion?

```javascript
function outer() {
  var x = 0;
  function inner() {
    x = x + 1;
    return x;
  }
  return inner;
}
```
JavaScript

Geschachtelte Funktionen?

Function-Call Frames und Rekursive Funktionen

Früher: Keine Unterstützung für rekursiven Funktionen (z.B. Fortran 77)

- Funktionen werden instanziierbar; jeder Aufruf erzeugt Call Frame
  - Ein Call-Frame hat einen Zeiger auf seinen aufrufenden Namensraum (caller).
  - Dynamische Namensauflösung erfolgt relativ zum Stackpointer (SP).

```
int fak(int n) {
  if (n < 1) {
    return 1;
  }

  int tmp = fak(n-1);

  return n * tmp;
}
```

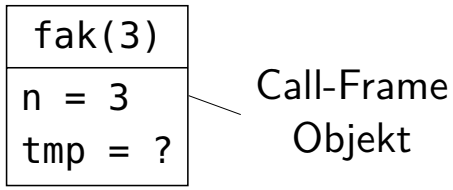
```
> fak(3);
```

Function-Call Frames und Rekursive Funktionen

Früher: Keine Unterstützung für rekursiven Funktionen (z.B. Fortran 77)

- Funktionen werden instanziierbar; jeder Aufruf erzeugt Call Frame
  - Ein Call-Frame hat einen Zeiger auf seinen aufrufenden Namensraum (caller).
  - Dynamische Namensauflösung erfolgt relativ zum Stackpointer (SP).

```
int fak(int n) {
  if (n < 1) {
    return 1;
  }

  int tmp = fak(n-1);

  return n * tmp;
}
```



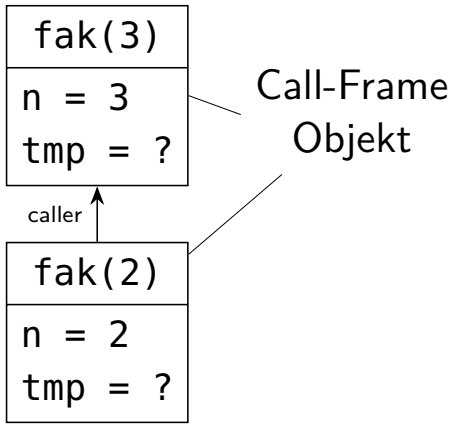

```
> fak(3);
```

Function-Call Frames und Rekursive Funktionen

Früher: Keine Unterstützung für rekursiven Funktionen (z.B. Fortran 77)

- Funktionen werden instanziierbar; jeder Aufruf erzeugt Call Frame
  - Ein Call-Frame hat einen Zeiger auf seinen aufrufenden Namensraum (caller).
  - Dynamische Namensauflösung erfolgt relativ zum Stackpointer (SP).

```
int fak(int n) {
  if (n < 1) {
    return 1;
  }

  int tmp = fak(n-1);

  return n * tmp;
}
```




```
> fak(3);
```

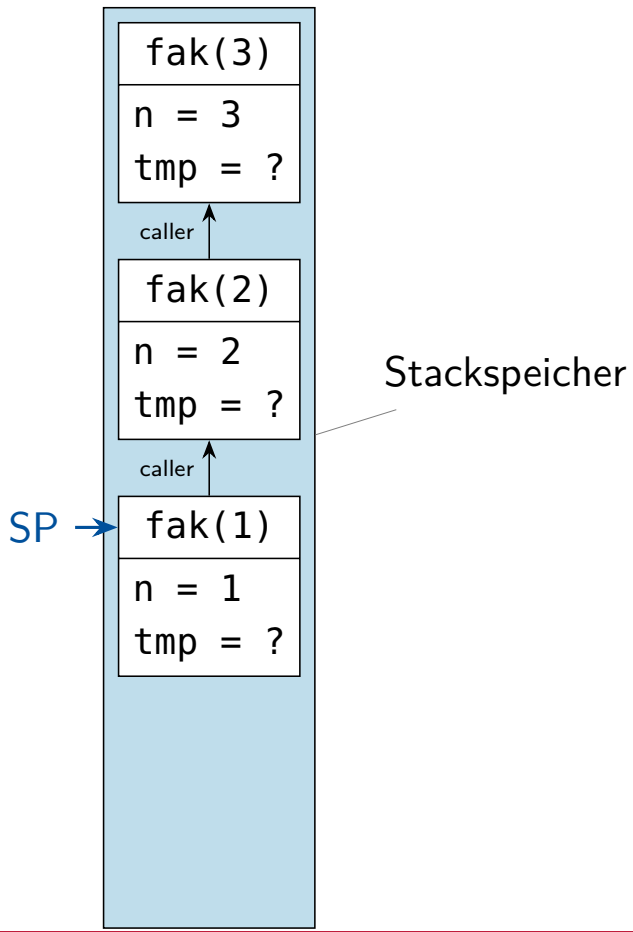
Function-Call Frames und Rekursive Funktionen

Früher: Keine Unterstützung für rekursiven Funktionen (z.B. Fortran 77)

- Funktionen werden instanziierbar; jeder Aufruf erzeugt Call Frame
  - Ein Call-Frame hat einen Zeiger auf seinen aufrufenden Namensraum (caller).
  - Dynamische Namensauflösung erfolgt relativ zum Stackpointer (SP).

```
int fak(int n) {
  if (n < 1) {
    return 1;
  }

  int tmp = fak(n-1);

  return n * tmp;
}
```

```
> fak(3);
```



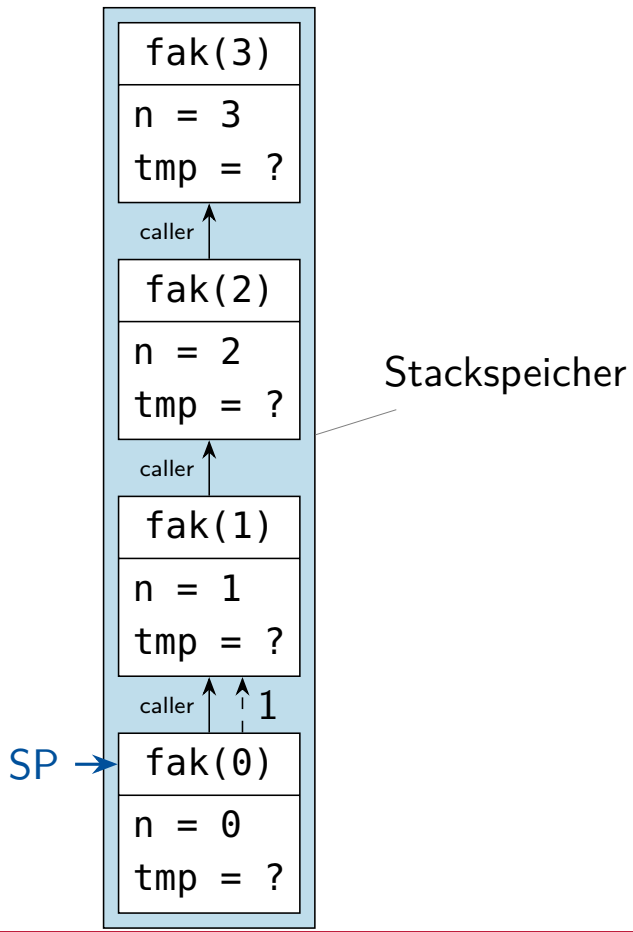
Function-Call Frames und Rekursive Funktionen

Früher: Keine Unterstützung für rekursiven Funktionen (z.B. Fortran 77)

- Funktionen werden instanziierbar; jeder Aufruf erzeugt Call Frame
  - Ein Call-Frame hat einen Zeiger auf seinen aufrufenden Namensraum (caller).
  - Dynamische Namensauflösung erfolgt relativ zum Stackpointer (SP).

```
int fak(int n) {
  if (n < 1) {
    return 1;
  }

  int tmp = fak(n-1);

  return n * tmp;
}
```

```
> fak(3);
```



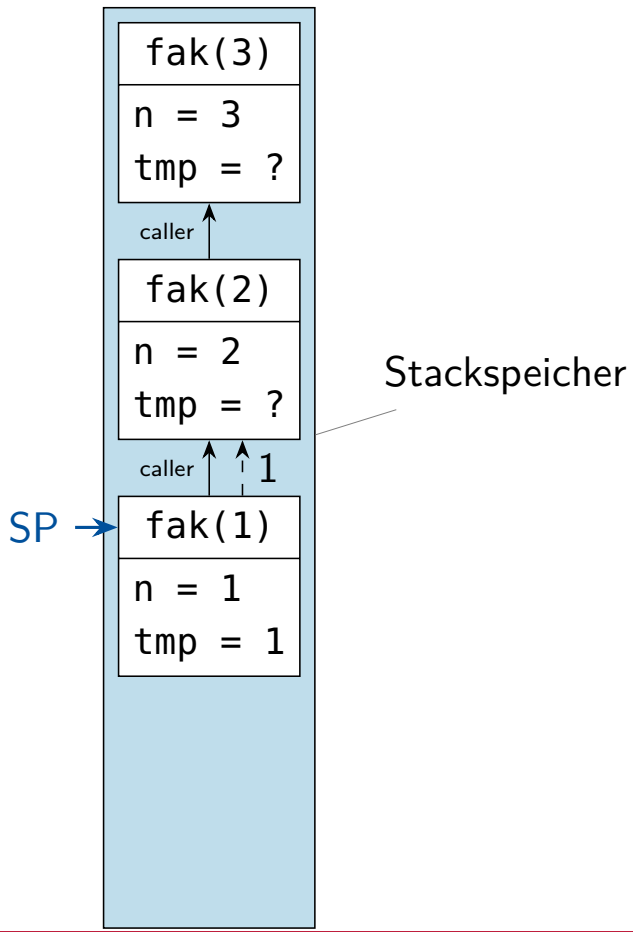
Function-Call Frames und Rekursive Funktionen

Früher: Keine Unterstützung für rekursiven Funktionen (z.B. Fortran 77)

- Funktionen werden instanziierbar; jeder Aufruf erzeugt Call Frame
  - Ein Call-Frame hat einen Zeiger auf seinen aufrufenden Namensraum (caller).
  - Dynamische Namensauflösung erfolgt relativ zum Stackpointer (SP).

```
int fak(int n) {
  if (n < 1) {
    return 1;
  }

  int tmp = fak(n-1);

  return n * tmp;
}
```

```
> fak(3);
```



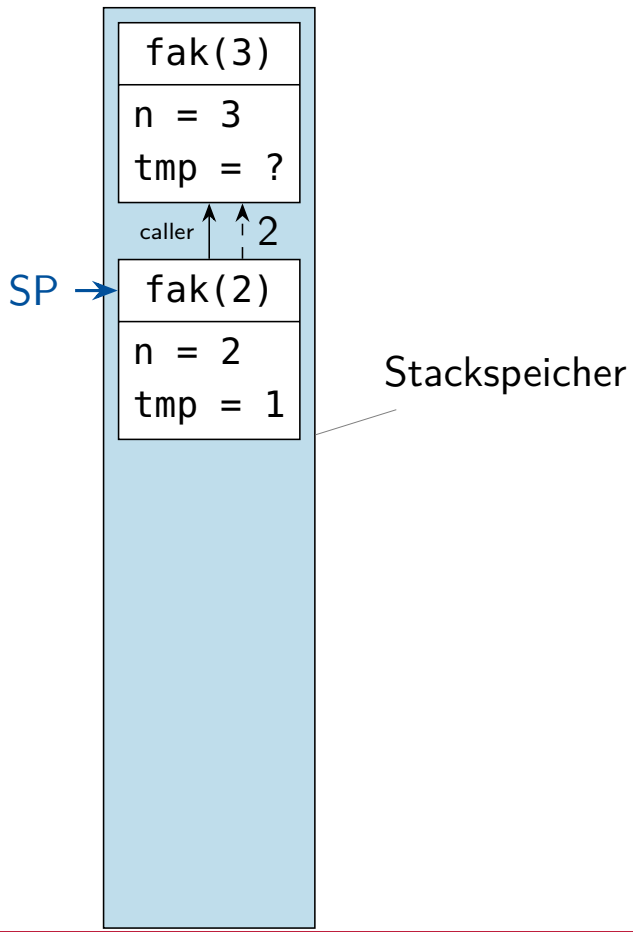
Function-Call Frames und Rekursive Funktionen

Früher: Keine Unterstützung für rekursiven Funktionen (z.B. Fortran 77)

- Funktionen werden instanziierbar; jeder Aufruf erzeugt Call Frame
  - Ein Call-Frame hat einen Zeiger auf seinen aufrufenden Namensraum (caller).
  - Dynamische Namensauflösung erfolgt relativ zum Stackpointer (SP).

```
int fak(int n) {
  if (n < 1) {
    return 1;
  }

  int tmp = fak(n-1);

  return n * tmp;
}
```

```
> fak(3);
```



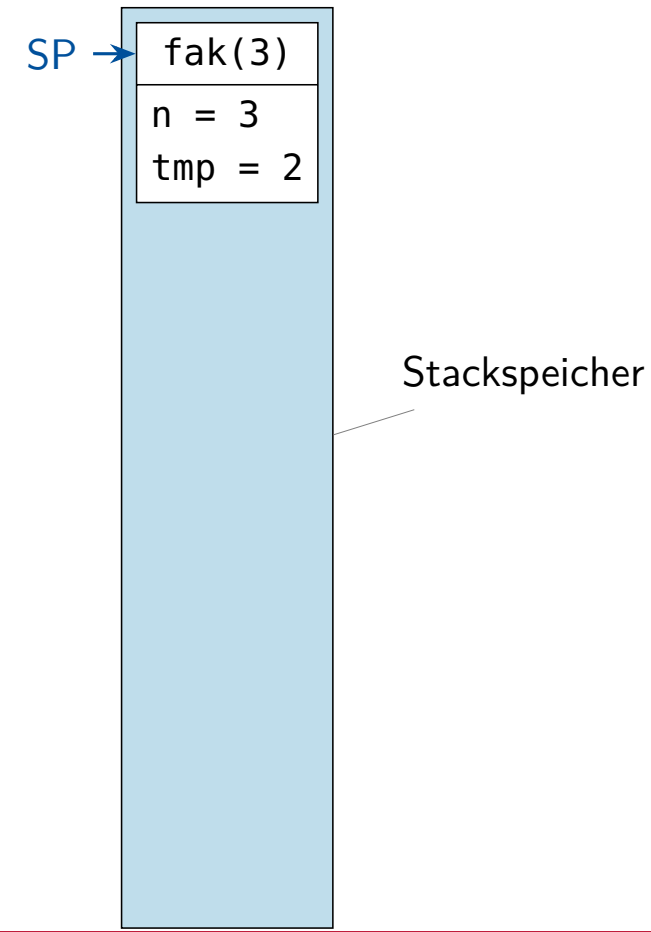
Function-Call Frames und Rekursive Funktionen

Früher: Keine Unterstützung für rekursiven Funktionen (z.B. Fortran 77)

- Funktionen werden instanziierbar; jeder Aufruf erzeugt Call Frame
  - Ein Call-Frame hat einen Zeiger auf seinen aufrufenden Namensraum (caller).
  - Dynamische Namensauflösung erfolgt relativ zum Stackpointer (SP).

```
int fak(int n) {
  if (n < 1) {
    return 1;
  }

  int tmp = fak(n-1);

  return n * tmp;
}
```

```
> fak(3);
```



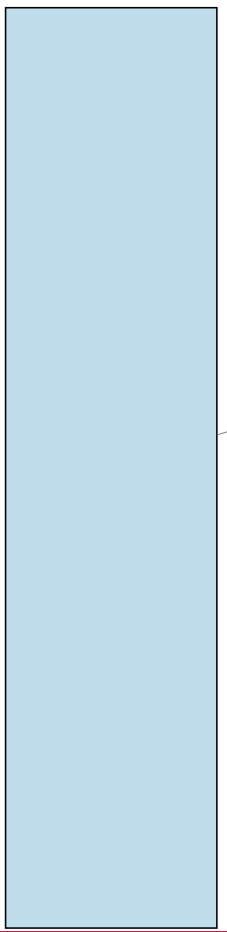
Function-Call Frames und Rekursive Funktionen

Früher: Keine Unterstützung für rekursiven Funktionen (z.B. Fortran 77)

- Funktionen werden instanziierbar; jeder Aufruf erzeugt Call Frame
  - Ein Call-Frame hat einen Zeiger auf seinen aufrufenden Namensraum (caller).
  - Dynamische Namensauflösung erfolgt relativ zum Stackpointer (SP).

```
int fak(int n) {
  if (n < 1) {
    return 1;
  }

  int tmp = fak(n-1);

  return n * tmp;
}
```

```
> fak(3);
6;
>
```



Geschachtelte Funktionen und Closures

- Funktionen in Funktionen lassen Fragen über die Bindung aufkommen.
  - Die Deklaration zu einer Referenz ist nach CNSR klar.
  - **Aber**: Wir brauchen einen `outer`-Call-Frame, um `x` vollständig aufzulösen.

```
function outer(init) {
  var x = init;
  function inner() {
    x = x + 1;
    return x;
  }
  return inner;
}
```

outer(10)
init = 10
x = 10
inner = ?

```
> var f = outer(10);
```

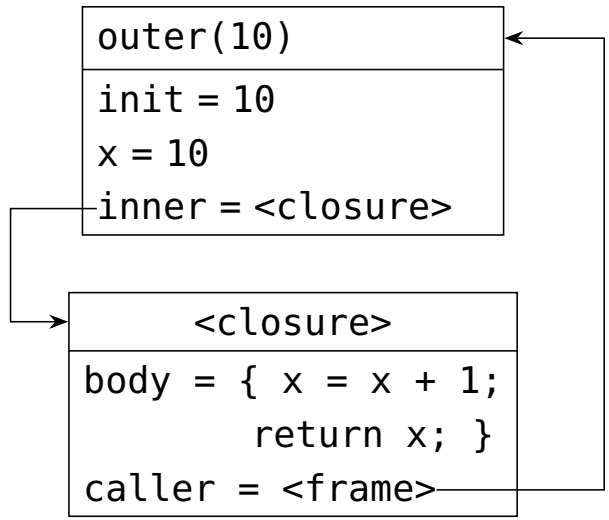
- Lexikalisches Scoping: Funktionen können Zustand halten
  - Bei der **Definition** von `inner` wird ein Closure-Objekt erstellt.
  - Die Closure speichert den Frame über die Ausführung von `outer()` hinaus.

Geschachtelte Funktionen und Closures

- Funktionen in Funktionen lassen Fragen über die Bindung aufkommen.
  - Die Deklaration zu einer Referenz ist nach CNSR klar.
  - **Aber**: Wir brauchen einen `outer`-Call-Frame, um `x` vollständig aufzulösen.

```
function outer(init) {
  var x = init;
  function inner() {
    x = x + 1;
    return x;
  }
  return inner;
}
```

```
> var f = outer(10);
```



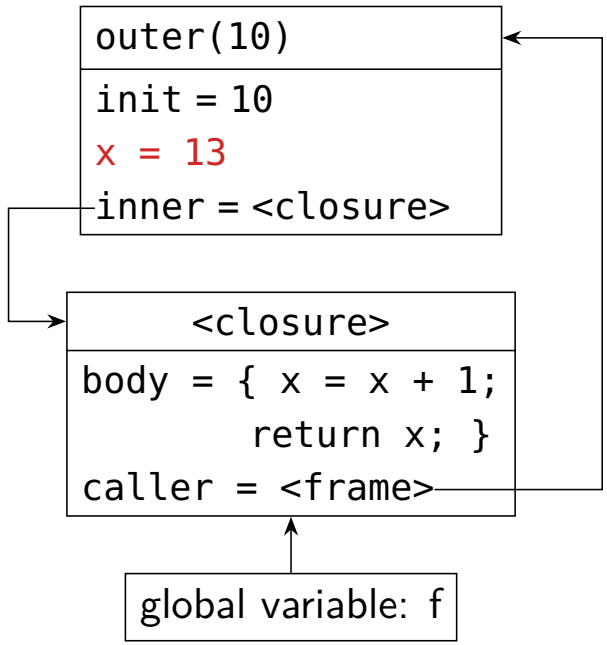
- Lexikalisches Scoping: Funktionen können Zustand halten
  - Bei der **Definition** von `inner` wird ein Closure-Objekt erstellt.
  - Die Closure speichert den Frame über die Ausführung von `outer()` hinaus.

Geschachtelte Funktionen und Closures

- Funktionen in Funktionen lassen Fragen über die Bindung aufkommen.
  - Die Deklaration zu einer Referenz ist nach CNSR klar.
  - **Aber**: Wir brauchen einen `outer`-Call-Frame, um `x` vollständig aufzulösen.

```
function outer(init) {
  var x = init;
  function inner() {
    x = x + 1;
    return x;
  }
  return inner;
}
```

```
> var f = outer(10);
<closure>
>[f(), f(), f()]
[11, 12, 13]
```



- Lexikalisches Scoping: Funktionen können Zustand halten
  - Bei der **Definition** von `inner` wird ein Closure-Objekt erstellt.
  - Die Closure speichert den Frame über die Ausführung von `outer()` hinaus.

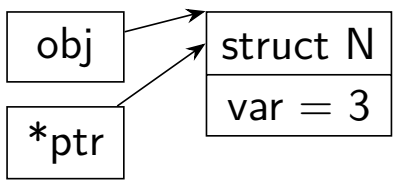
Pointer, Aliase und Uneindeutigkeit

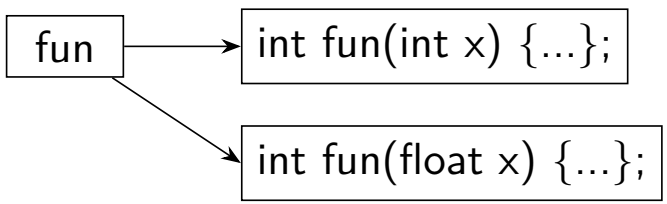
Wiederholung: Namensauflösung findet das Objekt zu einem Namen.

- Ein Zeiger speichert das Ergebnis einer vollständigen Namensauflösung.
  - Der `&(<Namenspfad>)`-Operator löst den Pfad auf und gibt eine Adresse zurück.
  - Der `(*<Name>)`-Operator macht das referenzierte Objekt zugänglich.

```c
struct N {
  int var;
};
struct N obj;
struct N* ptr = &obj;
// obj.var == (*ptr).var;
```

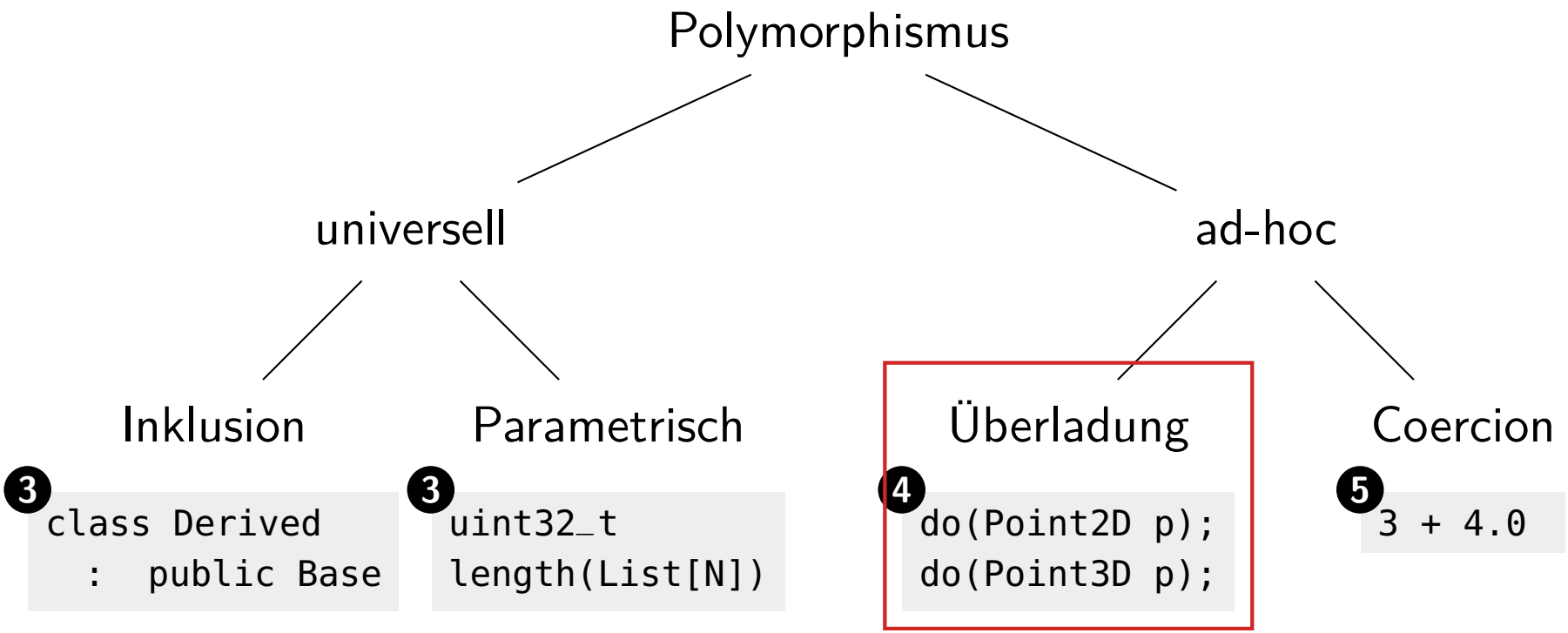
- Aliase: Mehrere Namen (`obj`, `*ptr`) binden nun das selbe Objekt.





Wäre es nützlich, wenn ein Name mehrere Objekte meinen könnte?

Erinnerung: Arten des Polymorphismus



Überladene Funktionen in a Nutshell

Technische Universität Braunschweig

Vorweg: Vollständige Namensauflösung ist immer eindeutig!

Das Ergebnis ist immer genau ein Objekt. Allerdings können wir die Berechnung, bis zur tatsächlich Verwendung, in die Laufzeit verschieben.

- Bei überladenen Funktionen werden die Parametertypen Teil des Namens

```C++
bool isNeg(int x);
bool isNeg(double x);
bool isNeg(string x);
```

```
let isNeg(int)
let isNeg(double)
let isNeg(string)
```

  - Alle Deklarationen sind weiterhin eindeutig.
  - Aber: Eine Referenz mit dem Namen **`isNeg`** ist (allein) nicht mehr eindeutig.
- Die Namensauflösung wird erst im konkreten Aufrufkontext eindeutig.

```C++
bool fn() {
  string value = "-3";
  return isNeg(value);
}
```

```
let fn{[I]
  let value // type=string
  isNeg(string)
}
```

Namensauflösung für überladene Funktionen

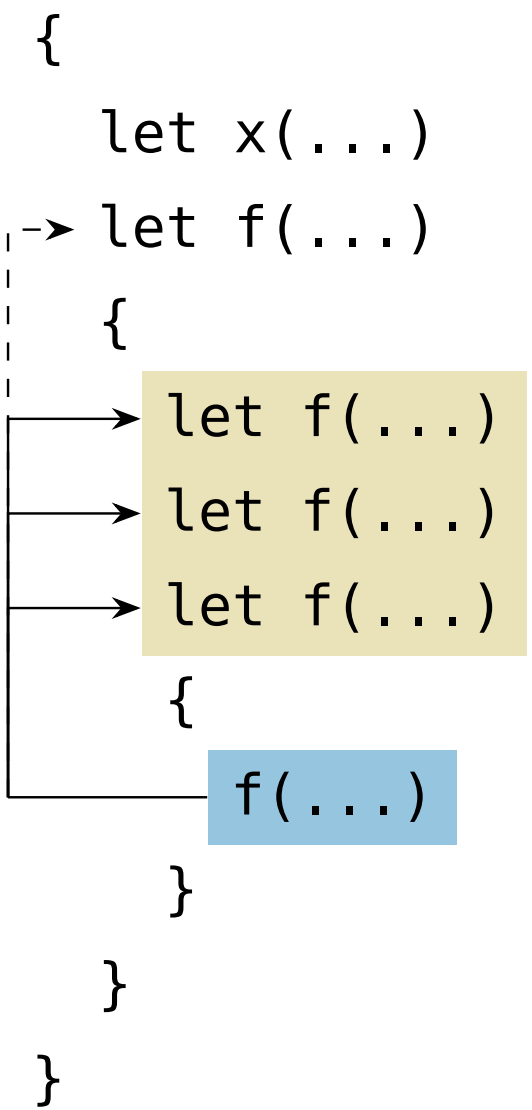
```
{
  let x(...)
  let f(...)
  {
    let f(...)
    let f(...)
    let f(...)
    {
      f(...)
    }
  }
}
```

Vorläufige Annahme

Monomorphes und statisches Typsystem

1. Overload-Set bestimmen
   - Suche nach sichtbaren Deklaration
   - Abbruch der Suche beim ersten Treffer

Namensauflösung für überladene Funktionen

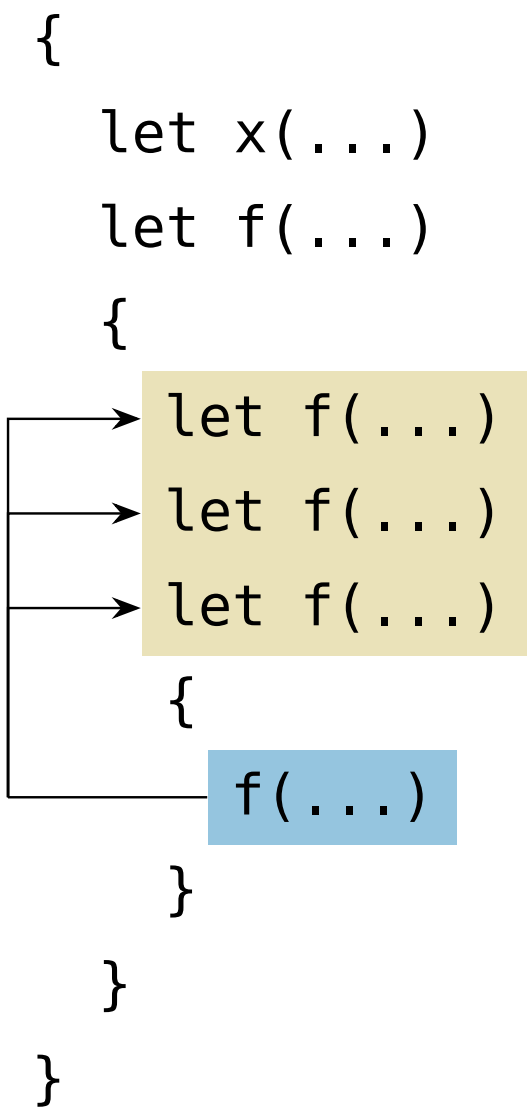


Vorläufige Annahme

Monomorphes und statisches Typsystem

1. Overload-Set bestimmen
   - Suche nach sichtbaren Deklaration
   - Abbruch der Suche beim ersten Treffer

Namensauflösung für überladene Funktionen



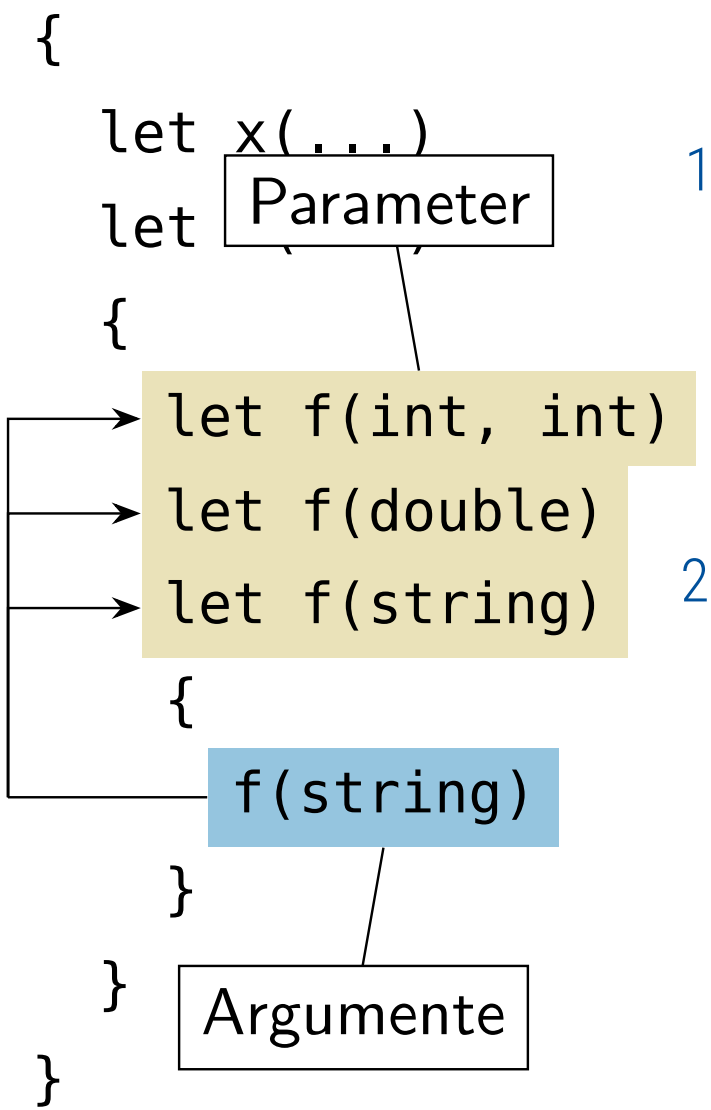
Vorläufige Annahme

Monomorphes und statisches Typsystem

1. Overload-Set bestimmen
   - Suche nach sichtbaren Deklaration
   - Abbruch der Suche beim ersten Treffer

Java: Überladung vs. Überschreiben

Namensauflösung für überladene Funktionen



Vorläufige Annahme

Monomorphes und statisches Typsystem

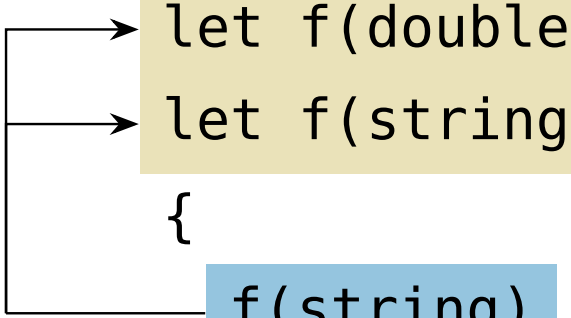
1. Overload-Set bestimmen
   - Suche nach sichtbaren Deklaration
   - Abbruch der Suche beim ersten Treffer

   Java: Überladung vs. Überschreiben

2. Typsignaturen durch Typsystem ermitteln
   - Deklaration und Aufrufstelle
   - Liste der Parameter und der Argumente

Namensauflösung für überladene Funktionen

```
{
  let x(...)
  let f(...)
  {
    let f(int, int)
    let f(double)
    let f(string)
    {
      f(string)
    }
  }
}
```



Vorläufige Annahme

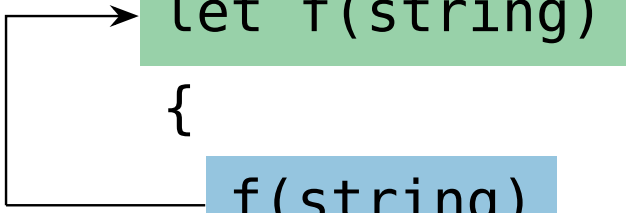
Monomorphes und statisches Typsystem

1. Overload-Set bestimmen
   - Suche nach sichtbaren Deklaration
   - Abbruch der Suche beim ersten Treffer

   Java: Überladung vs. Überschreiben
2. Typsignaturen durch Typsystem ermitteln
   - Deklaration und Aufrufstelle
   - Liste der Parameter und der Argumente
3. Einschränkung des Overload-Sets
   - Übereinstimmung in Parameteranzahl
   - Typen müssen gleich sein

Namensauflösung für überladene Funktionen

```
{
  let x(...)
  let f(...)
  {
    let f(int, int)
    let f(double)
    let f(string)
    {
      f(string)
    }
  }
}
```



Vorläufige Annahme

Monomorphes und statisches Typsystem

1. Overload-Set bestimmen
   - Suche nach sichtbaren Deklaration
   - Abbruch der Suche beim ersten Treffer

   Java: Überladung vs. Überschreiben
2. Typsignaturen durch Typsystem ermitteln
   - Deklaration und Aufrufstelle
   - Liste der Parameter und der Argumente
3. Einschränkung des Overload-Sets
   - Übereinstimmung in Parameteranzahl
   - Typen müssen gleich sein

   Statische Namensauflösung möglich!

Fallbeispiel: Statische Überladung in C++

```C++
bool isNeg(int x)    {...}
bool isNeg(double x) {...}
bool isNeg(int x, int y) {...}

bool fn() {
  return isNeg(2.0);
}
```

- Name Mangling: Die Parameterliste wird in die Assemblernamen codiert
- Der Übersetzer löst `isNeg(2.0)` statisch zu `isNeg(double)` auf.
- `call` ruft einen „mangled name“ auf; der Linker fügt die richtige Adresse ein.

Symbole im ELF

```
> nm a.out | grep isNeg
....
00000000000011e4 T _Z5isNegd
00000000000011d5 T _Z5isNegi
0000000000001222 T _Z5isNegii
....
```

Assembler Code

```
_Z3fnv:
   push   %rbp
   mov    %rsp,%rbp
   mov    0xd0c(%rip),%rax
   movq   %rax,%xmm0
   callq  11e4 <_Z5isNegd>
   pop    %rbp
   retq
```

Operatorüberladung

- Arithmetische Operatoren (`+`, `-`, `*`, `/`) sind überladen
  - Interpretation als Funktionsaufrufe: `2 + 3` → `+(2, 3)`
  - Der Operator (z.B. `+` oder `++`) wird zum Funktionsnamen: `++(a)`
  - Sprachen deklarieren bereits mehrere überladene Implementierungen:

    `+ :: (int, int) ↦ int` — Addition von Ganzzahlen

    `+ :: (float, float) ↦ float` — Addition von Gleitkommazahlen

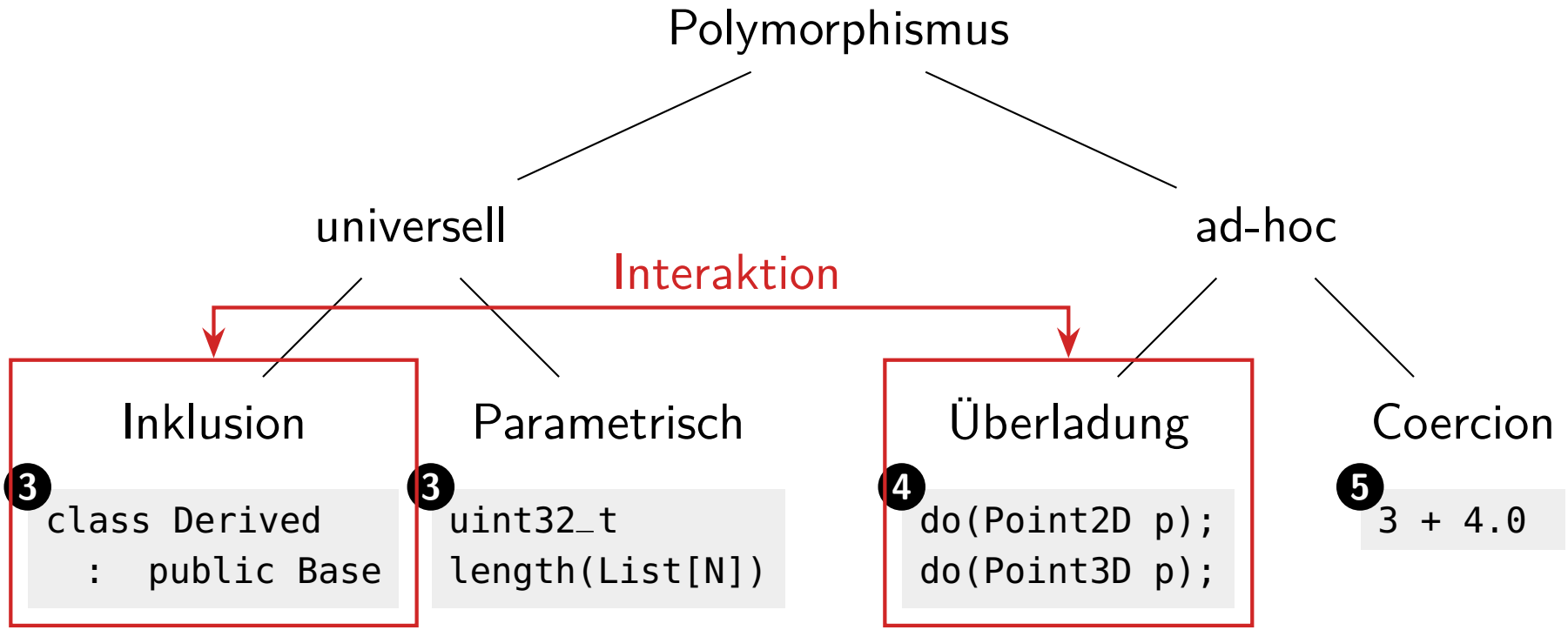
    `+ :: (string, string) ↦ string` — Konkatenierung von Zeichenketten

  ⇒ Der Übersetzer braucht bereits Infrastruktur zur Überladungsauflösung

- Manche Sprachen erlauben eigene Operatorüberladungen
  - Die selbst-deklarierte Überladungen müssen „magischen“ Namen tragen
  - Overload-Set enthält eingebaute und selbst-deklarierten Operatoren

```C++
struct complex {
  int real, imag;
  complex(int r, int i)  {real = r;   imag = i;} // Konstruktor
};

complex operator+ (complex c1, complex c2) {
  return complex(c1.real + c2.real, c1.imag + c2.imag);
}
```

Erinnerung: Arten des Polymorphismus



Interaktion zwischen Überladung und Subtypen

Erinnerung: Subtyping ist eine Form des universellen Polymorphismus. Bei Vererbung ist die Kindklasse ein Subtyp der Elternklasse.

Ein polymorphes Typsystem führt Typkompatibilität ein und erzeugt damit Uneindeutigkeit. Diese muss die Sprache eindeutig auflösen.

An der Deklarationsstelle

```C++
class S : public B {}
void f(B * obj) {...}
void f(S * obj) {...}
...
S * obj = new S();
f(obj); // f(S *)
```

- Subtyp `S` kompatibel zu `B`
- Beide Dekls.. passen zum Aufruf
- Welche Deklaration wählen wir?

⇒ Berechnung der Spezifität

An der Aufrufstelle

```C++
// wie links
void f(B * this, int x) {...}
void f(S * this, int x) {...}
...
B *obj = new S();
f(obj, 23);
```

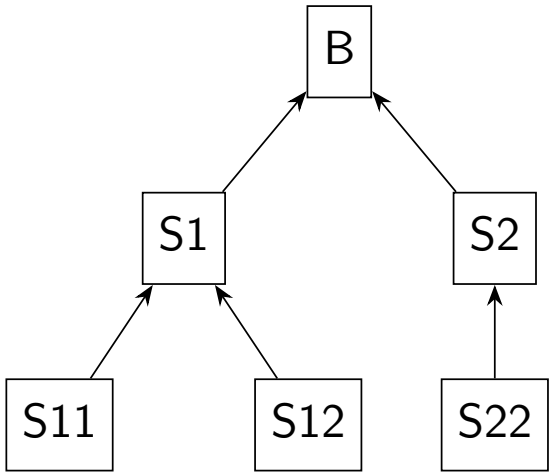
- Variablendeklaration vs. Objekt
- Statischer Typ != Dynamischer Typ
- Welchen Typen sollen wir beachten?

⇒ Dynamischer Dispatch

Auswahl der spezifischsten Überladung

Technische Universität Braunschweig

Vererbungshierarchie

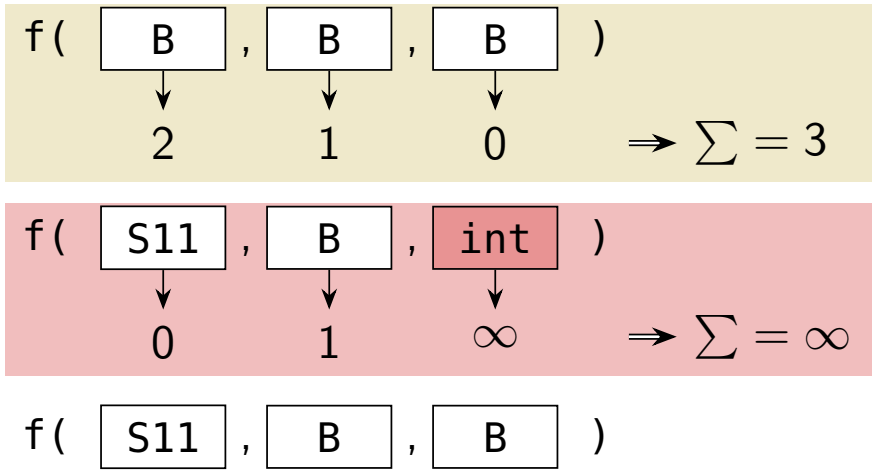


Aufrufstelle

```
f( S11 , S2, B)
```

Overload Set

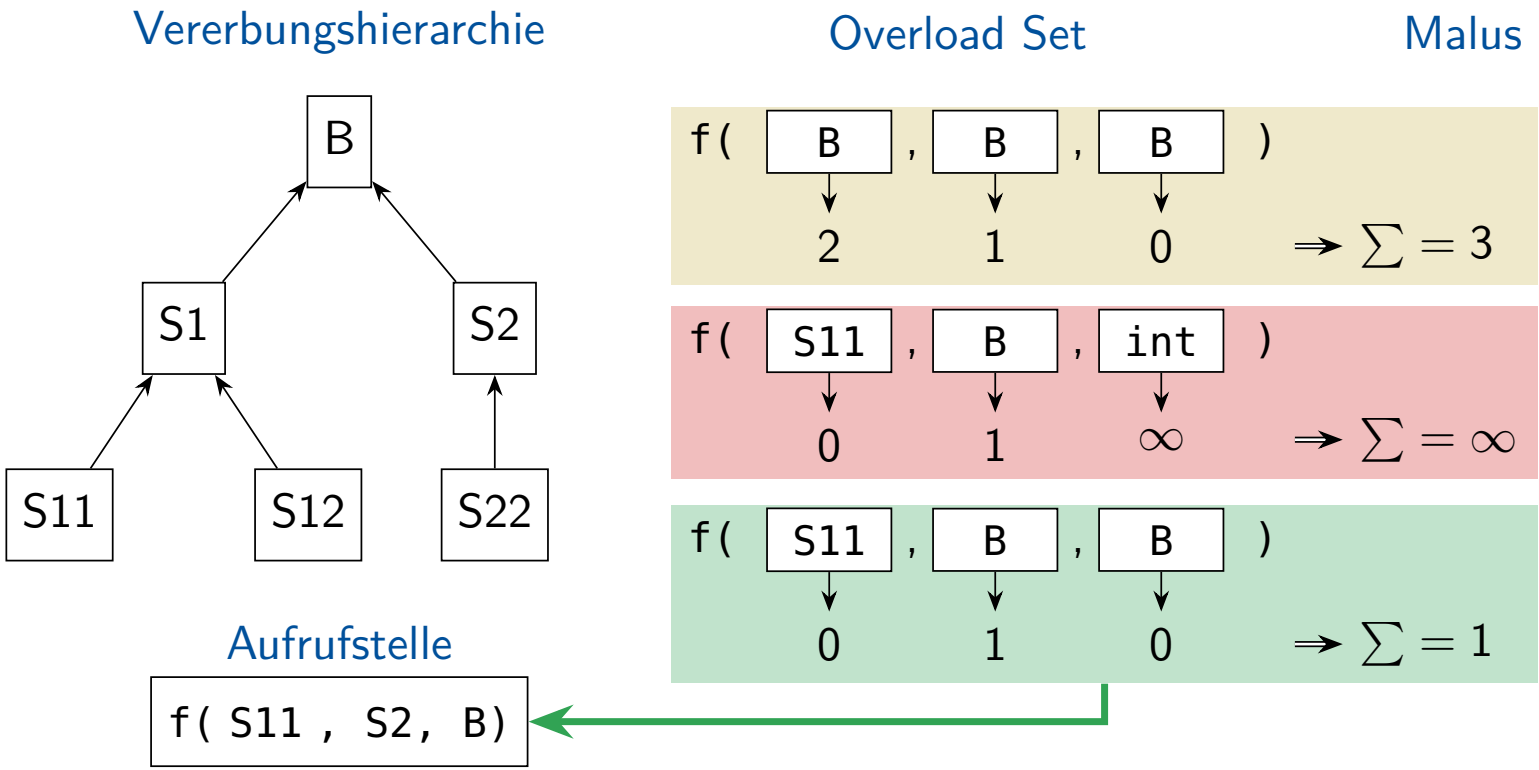
Malus



Intuition: Umso mehr Casts notwendig sind, umso schlechter die Spezifität.

- Bestimmung der Argument–Parameter Distanz in der Vererbungshierarchie
- Inkompatible Typen erzeugen eine Distanz von ∞
- Der niedrigste Malus ist die höchste Spezifität

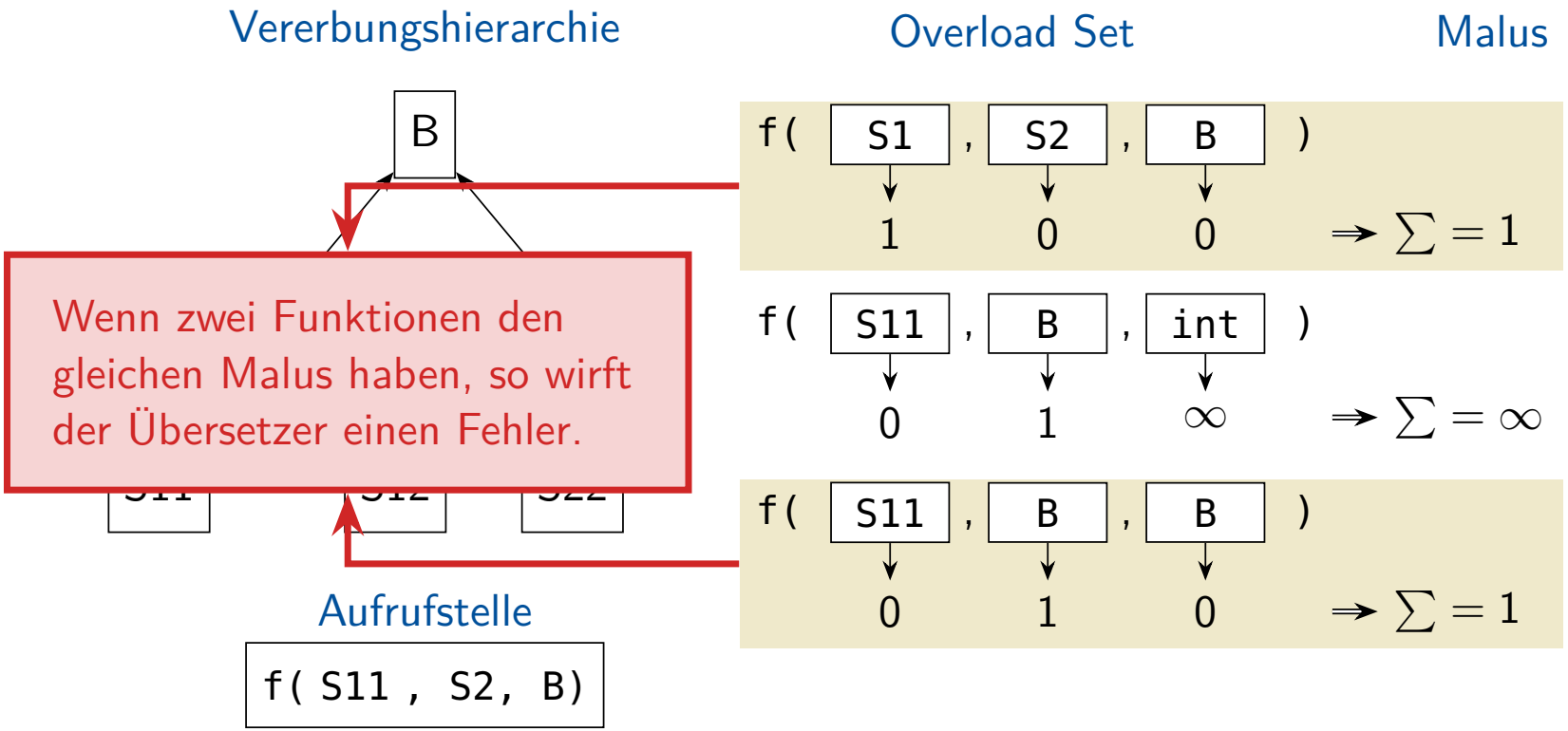
Auswahl der spezifischsten Überladung



Intuition: Umso mehr Casts notwendig sind, umso schlechter die Spezifität.

- Bestimmung der Argument–Parameter Distanz in der Vererbungshierarchie
- Inkompatible Typen erzeugen eine Distanz von ∞
- Der niedrigste Malus ist die höchste Spezifität

Auswahl der spezifischsten Überladung



Intuition: Umso mehr Casts notwendig sind, umso schlechter die Spezifität.

- Bestimmung der Argument–Parameter Distanz in der Vererbungshierarchie
- Inkompatible Typen erzeugen eine Distanz von ∞
- Der niedrigste Malus ist die höchste Spezifität

Dynamischer Dispatch anhand des dynamischen Typs

```java
Base obj = new Derived();
func(obj);
// dynamisch: func(Derived)
// statisch:  func(Base)
```

Java

```python
def f(obj):
    if isinstance(obj, Derived):
        return f_Derived(obj)
    elif isinstance(obj, Base):
        return f_Base(obj)
    raise TypeError()
```

Py

Bisher haben wir nur die statischen Typen der Argumente betrachtet.

- Dynamic Dispatch: Überladungsauswahl anhand des dynamischen Typs
  1. Vorberechnung eines eingeschränkten Overload-Sets
  2. Zur Laufzeit: Extraktion des dynamischen Typs aller Argumente
  3. Berechnung und Aufruf der spezifischsten Überladung
- Multiple Dynamic Dispatch: Dynamischer Dispatch allen Argumenten
  - Wird nur von wenige Sprachen direkt unterstützt (Common Lisp, Julia)
  - Auswahl der passendsten Variante erzeugt hohen Aufwand zur Laufzeit
  - Emulation mit Hilfskonstrukten, wie `isinstance` Kaskaden.

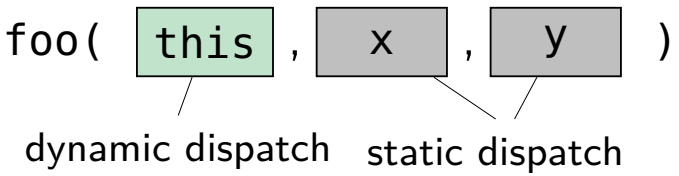
⇒ Eingeschränkte Form (single dynamic dispatch) wird häufig angeboten.

Single Dynamic Dispatch: Virtuelle Methoden

Virtuelle Methoden sind eine Form von Dynamic Dispatch.

```java
class Base {
  public void foo(int x, int y){...}
}
class Derived extends Base {
  public void foo(int x, int y){...}
}
...
Base obj = new Derived();
obj.foo(1, 1);
```

Java



- Methoden werden im Kontext einer Klasse deklariert.
  - Methoden werden „auf einem Objekt“ aufgerufen: `obj.foo(1, 1)`
  - Das Objekt wird zu einem impliziten Argument (`this := obj`)
  - Bei statischen Methoden wird nur der statische Typ beachtet.
- Virtuelle Methoden: dynamischer Dispatch auf dem `this`-Argument
  - Die passende Überladung hängt am Objekt, nicht an der Variable.
  - Effizient implementierbar mit virtuellen Funktionstabellen (vtable).
  - In Java ist jede Methode, per default, eine virtuelle Methode.

Zusammenfassung

- Namen sind symbolische und statische Objektreferenzen.
  - Bei der Deklaration können Namen mit (Typ-)informationen versehen werden.
  - Deklarationen werden in hierarchischen Namensräumen strukturiert.
  - Eingeschränkte Sichtbarkeit und Import von Namen möglich
- Namensauflösung berechnet aus einem Namen die Objektadresse
  - Vollständige Auflösung ist nur nach der Binding Time eines Namens möglich
  - Partielle Auflösung kann bereits vorgezogen werden (Variablendeklaration)
- Instanziierbare Namensräume führen zu dynamischer Namensauflösung
  - Eine Deklaration beschreibt viele Laufzeitobjekte (`struct`, `class`).
  - Laufzeitwissen wird zur vollständigen Namensauflösung benötigt (`this`).
- Überladene Funktionen werden mit Typinformationen aufgelöst
  - Neben dem Funktionsnamen wird auch die Typsignatur einbezogen.
  - Bei Subtyping: Auswahl zwischen statischem und dynamischem Dispatch