Heterogenes Typaggregat: Record

- Ausrichtung der Komponenten: (alignment)
  - Ausrichtungseinschränkungen führen u.U. zu Speicherlücken
  - Bsp.: Sun Sparc Prozessor
    - `int` auf 4, `double` auf 8 Byte ausgerichtet
    - `struct ix {int i; double x}` braucht 16 Bytes (statt 12)
  - Alignmentanforderung:
    kgV aller Alignments (bei Zweierpotenzen: Maximum)
- Auswahl einer Komponente:
  Berechnung seiner Adresse und Dereferenzierung
- Kopie eines Records: Kopie seines Speicherbereichs
- Vergleich von Records: komponentenweise (ohne Lücken mit mglw. unterschiedlich belegten Bitmustern)

Typvereinigung: Union

- Union-Variablen können Werte unterschiedlichen Typs haben
  - Auswahl per Komponentennamen, wie beim Record
  - Bsp. in C: `union { int i; double x } a;`

    `a` enthält entweder `i` oder `x`
- Undiscriminated Union (in C)
  - nicht typsicher: Folge `a.i=n;d=a.x;` möglich
  - Information über Fall muss getrennt gehalten werden
  - Speicherbedarf: Maximum des Bedarfs aller Alternativen
- Discriminated Union (Haskell: `data U = I Int | D Double`)
  - implementiert als Paar (Typfeld (tag), Datenfeld)
  - Interpretation des Datenfeldes nach Angabe im Typfeld

    (oft vor der Laufzeit resolvierbar)

Homogenes Typaggregat: Array

- zwei Mengen:
  - Indexmenge (oft Integer) und
  - Wertmenge (Einträge im Array)
- Speicherverbrauch: Vielfaches des Verbrauchs eines Eintrags (keine Lücken zwischen Elementen)
- Dimensionalität
  - eindimensional: Vektor (C, Java, Pascal, Fortran)
  - zweidimensional: Matrix (Pascal, Fortran, C seit C99)
- Besonderheiten in C
  - Name eines Arrays ist zugleich Zeiger auf erstes Element
  - Addition auf Zeigern: `*(a+i)` gleich `a[i]` (`i` gewichtet)
  - bei Arraydurchlauf häufig Zeiger(in/de)kremente: `*(a++)`
  - Matrix ungleich Vektor von Vektoren (Syntax ähnlich!)

Speicherung einer Matrix

- blockweise Abbildung auf eindimensionalen Speicher
  (a) row-major order: erster Index gibt Block an (Pascal, C99)
  (b) column-major-order: zweiter Index gibt Block an (Fortran)

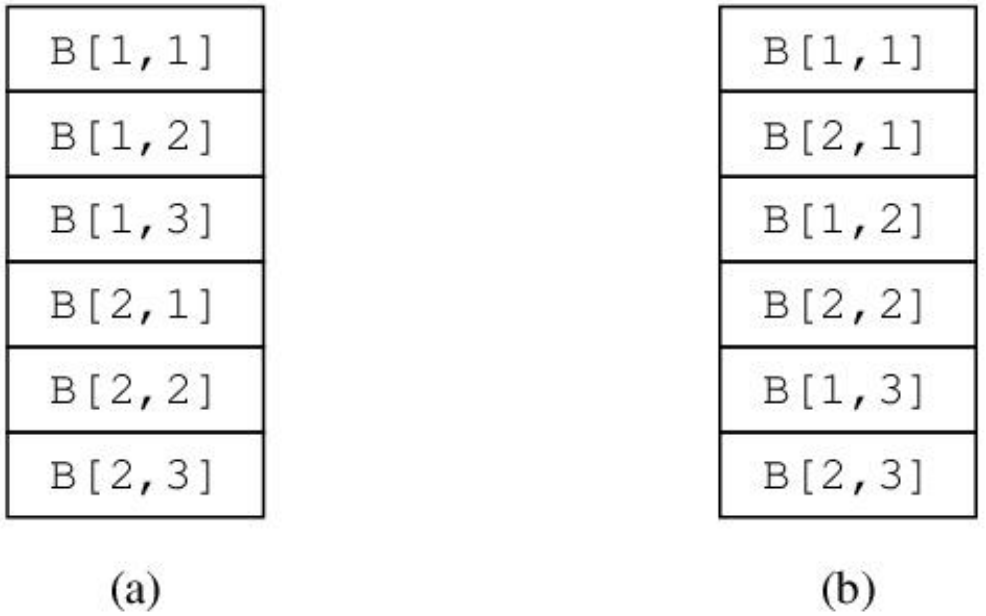


Figure 6.15 Array `B` in row-major (a) and column-major (b) order.

Speicherabbildung n-dimensionaler Arrays

- gegeben:
  - $base(A)$ (Anfangsadresse des Arrays A im Speicher)
  - el_size (Speicherbedarf für ein Arrayelement)
  - UB_r / LB_r (obere/untere Grenze in Dimension r, $1 \le r \le n$)

- Berechnungen zur Übersetzungszeit:

(a) $LEN_r = UB_r - LB_r + 1$ (Ausdehnung in Dimension k)

(b) $LEN\_PRODUCT_k = el\_size * \prod_{r=k+1}^{n} LEN_r$ (Speicherbedarf für Arrayblock ab Dimension k)

(c) $zeroth\_element(A) = base(A) - \sum_{k=1}^{n} LB_k * LEN\_PRODUCT_k$

(Adresse von evtl. fiktivem Element an Index [0,...,0])

- zur Laufzeit bleibt übrig zu berechnen:

$$address(A[i_1, \ldots, i_n]) = zeroth\_element(A) + \sum_{k=1}^{n} i_k * LEN\_PRODUCT_k$$

Array-Deskriptor (für die Codegenerierung)

- Adressberechnung $(zeroth\_element, LEN\_PRODUCT_k (1 \le k \le n))$
- Bound-Checking $(LB_r, LEN_r)\ (1 \le r \le n)$

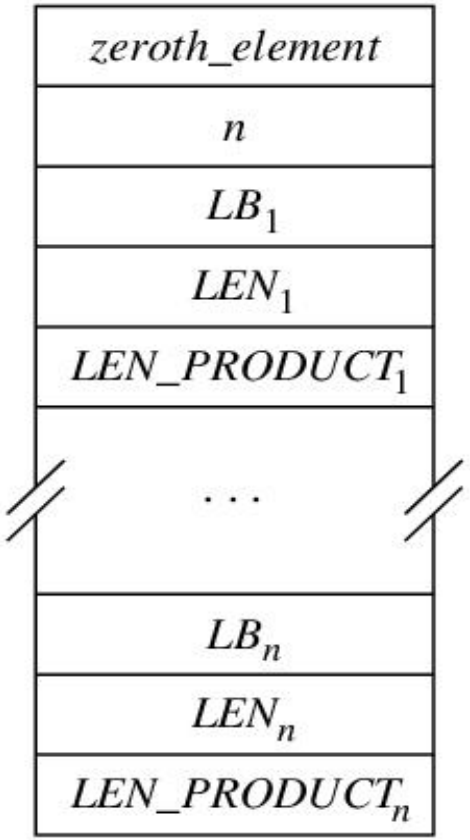


Figure 6.16 An array descriptor.

Mengentypen

- kleine, endliche Mengen: (`set` in Pascal)
  - repräsentiert als Bitvektor: ein Bit pro Element
  - i.d.R. Kardinalität ≤ Wortlänge
  - Operationen = Maschinenoperationen
    z.B. Vereinigung ≙ bitweisem logischen Oder
- unbeschränkte, endliche Mengen: (`set` in SETL)
  - repräsentiert als ungeordnete, verkettete Liste
- Dictionaries (in Smalltalk, C++, Java)
  - repräsentiert durch Suchbäume
- potenziell unendliche Mengen (in Haskell)
  - gegeben durch charakteristische Funktion

Repräsentation von Objekttypen

Implementierung folgender Features:

0. Zugriff auf Felder und Methoden
1. Vererbung
2. Overriding von Methoden
3. Subtyp-Polymorphie
4. Dynamische Bindung
5. Mehrfachvererbung
6. Abhängige Mehrfachvererbung

Zugriff auf Felder und Methoden

Bsp.: Klasse `A` mit Feldern `a1, a2` und Methoden `m1, m2`

- Laufzeitinformation: Record mit `a1` und `a2`
- Tabelle zur Übersetzungszeit enthält `m1_A` und `m2_A`
- Methode `m2_A` (ein `int`-Argument) als C-Routine:

```
void m2_A(Class_A *this, int i) {
      Body of method m2_A, accessing any
      object field x as this->x
   }
```

- Aufruf `a.m2(3)`:

```
void m2_A(&a,3)
```

Feature 1: Vererbung

Bsp.: Klasse `A` mit Feldern `a1`, `a2` und Methoden `m1`, `m2`

- Klasse `B` erweitert `A` um Feld `b1` und Methode `m3`
- Laufzeitinformation für Klasse `B`: Record mit `a1`, `a2`, `b1`
- Tabelle zur Übersetzungszeit enthält `m1_A`, `m2_A`, `m3_B`
- Implementierung wie zuvor

Feature 2: Overriding von Methoden (1)

Bsp.:

- Klasse `A` mit Feldern `a1`, `a2` und Methoden `m1`, `m2`
- Klasse `B` erweitert `A` um Feld `b1` und Methode `m3`
- Klasse `B` definiert eine neue Methode `m2`, die sich für `B`-Objekte gegenüber der in `A` definierten `m2`-Methode durchsetzt (override)

Notation `mN_Y_Z`:

Methode `N` ist (zuerst) *deklariert* in Klasse `Y` und *definiert* in Klasse `Z`

Methodentabelle

- für `A`: `m1_A_A`, `m2_A_A`
- für `B`: `m1_A_A`, `m2_A_B`, `m3_B_B`

Feature 2: Overriding von Methoden (2)

- Methodentabelle
  - für `A`: `m1_A_A`, `m2_A_A`
  - für `B`: `m1_A_A`, `m2_A_B`, `m3_B_B`
- Übersetzung der Aufrufe von `m2`
  - `a` Objekt der Klasse `A`: `a.m2(x)→m2_A_A(&a,x)`
  - `b` Objekt der Klasse `B`: `b.m2(x)→m2_A_B(&b,x)`
- Implementierung von `m2`

```
void m2_A_A(Class_A *this, int i) { ... };
void m2_A_B(Class_B *this, int i) { ... };
```

Feature 3: Subtyp-Polymorphie

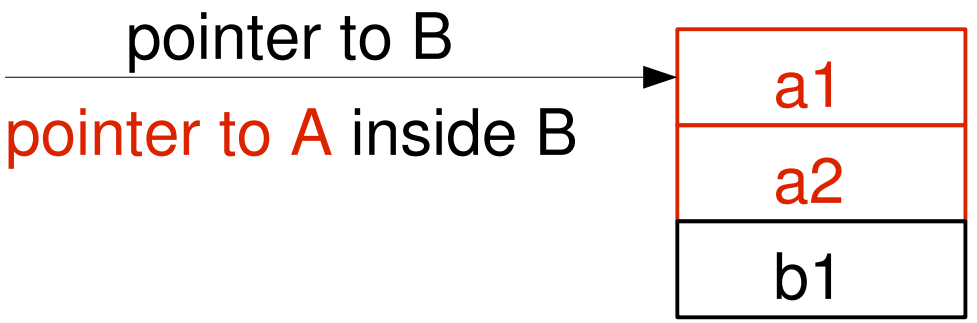
- Klasse `B` erweitert Klasse `A`
- Zuweisung (pointer to class `B`) an (pointer to class `A`)

```
class_B *b; b = ...;

class_A *a; a = b;
```

- Implementierung durch Pointer-Supertyping

```
class A *a;

a = convert_ptr_to_B_to_ptr_to_A(b);
```

- Pointer-Konversion bis jetzt reiner Compilezeit-Aspekt



Feature 4: Dynamische Bindung (1)

- Klasse `B` erweitert Klasse `A`, definiert Methode `m2` neu
- Pointer auf Klasse `A` kann auch auf Objekt der Klasse `B` zeigen
- Bezug von `a.m2()`?

  (a) statische Bindung: ➔ `m2_A_A`

  (b) dynamische Bindung

  (i) `a` Objekt der Klasse A: ➔ `m2_A_A`

  (ii) `a` Objekt der Klasse B: ➔ `m2_A_B`

  ⇒ dynamische Typinformation in der Objektrepräsentation

  ⇒ Pointer (re)subtyping zum Aufruf von `m2_A_B`

Feature 4: Dynamische Bindung (2)

Bsp.: `p` Zeiger (auf Objekttyp `A`), Objekt vom Typ `B`

Übersetzung des Methodenaufrufs `p->m2(3)`

```
switch (dynamic_type_of(p)) {
  case Dynamic_class_A:
     m2_A_A(p,3);
     break;
  case Dynamic_class_B:
     m2_A_B(convert_ptr_to_A_to_ptr_to_B(p),3);
     break;
}
```

Feature 4: Dynamische Bindung (3)

- einfacher: Integration der Konversion in Methodenrumpf

```
void m2_A_B(Class_A *this_A, int i) {
  Class_B *this;
  this = convert_ptr_to_A_to_ptr_to_B(this_A);
  Body of method m2_A_B, accessing any object field
  x as this->x
}
```

- Übersetzung des Methodenaufrufs `p->m2(3)`

```
(dynamic_type_of(p)==Dynamic_class_A ? m2_A_A : m2_A_B)
    (p, 3)
```

- Optimierung: anstelle der Abfrage speichere die Adresse der Funktion in der dynamischen Typinformation

Feature 4: Dynamische Bindung (4)

dynamische Typinformation enthält Zeiger auf Dispatch-Tabelle mit allen Methoden der Klasse des Objekts

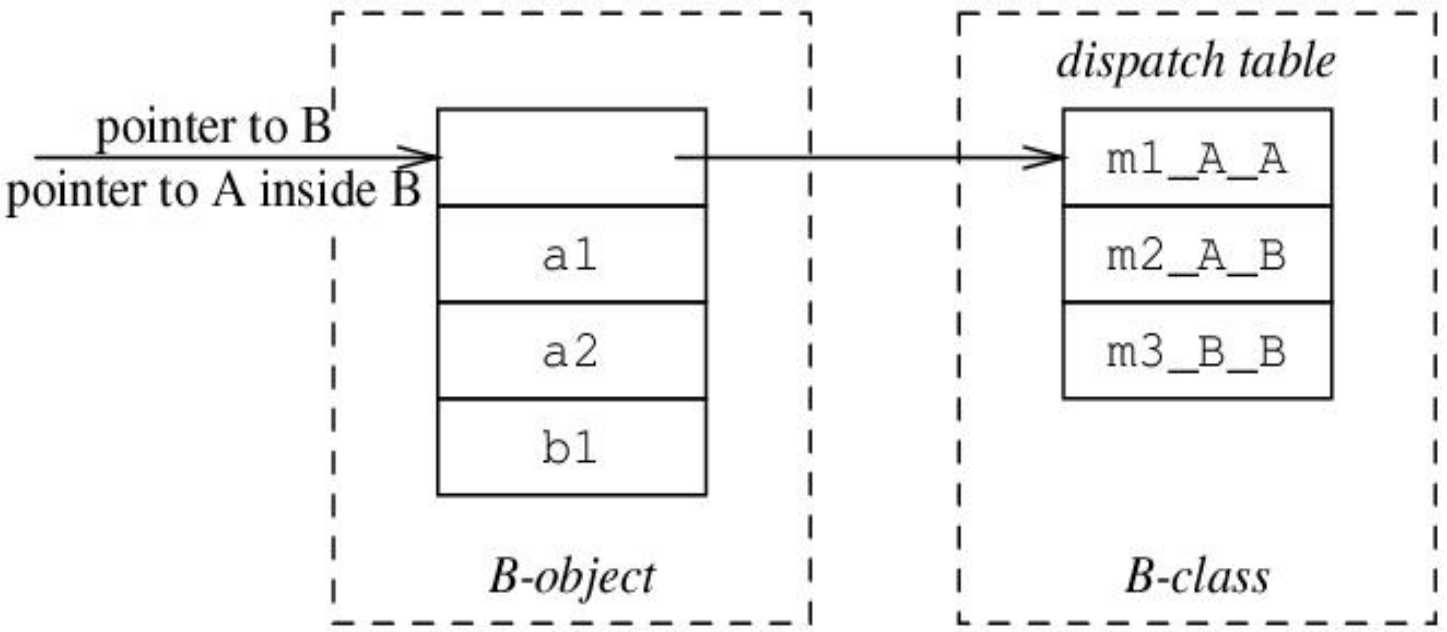
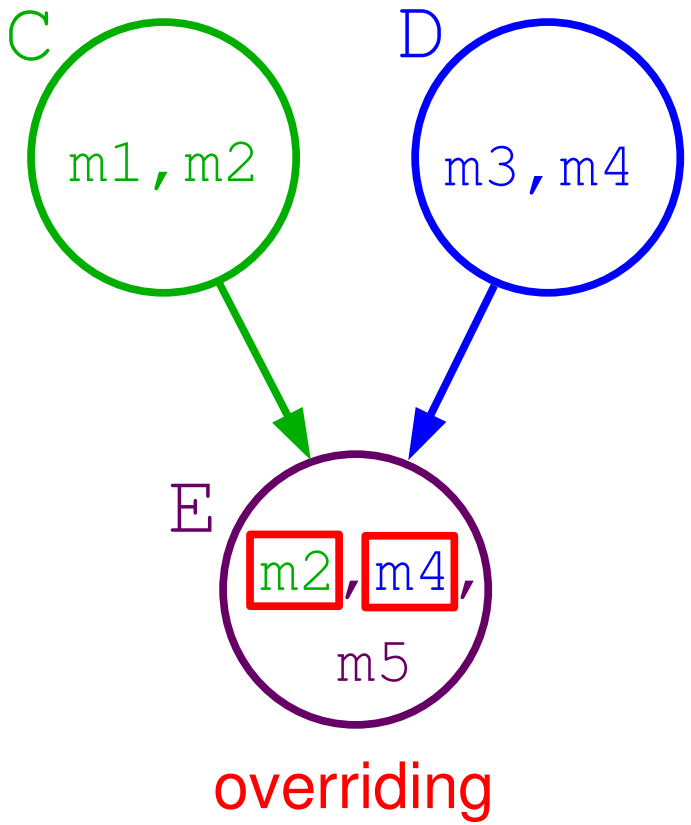


Figure 6.17 The representation of an object of class `B`.

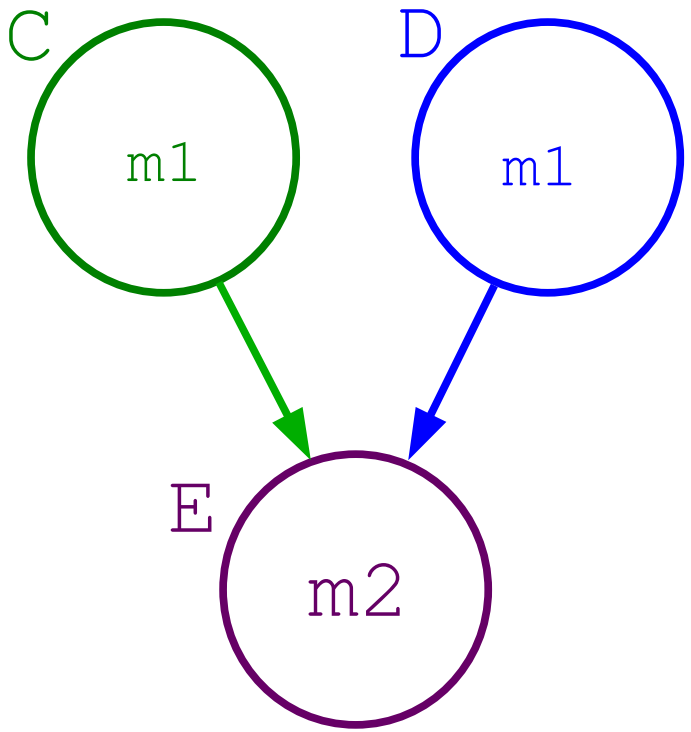
Feature 5: Mehrfachvererbung (1)




```
class C {
    field c1;
    field c2;
    method m1();
    method m2();
};

class D {
    field d1;
    method m3();
    method m4();
};

class E extends C, D {
    field e1;
    method m2();
    method m4();
    method m5();
};
```

Figure 6.18 An example of multiple inheritance.

Feature 5: Mehrfachvererbung (2)

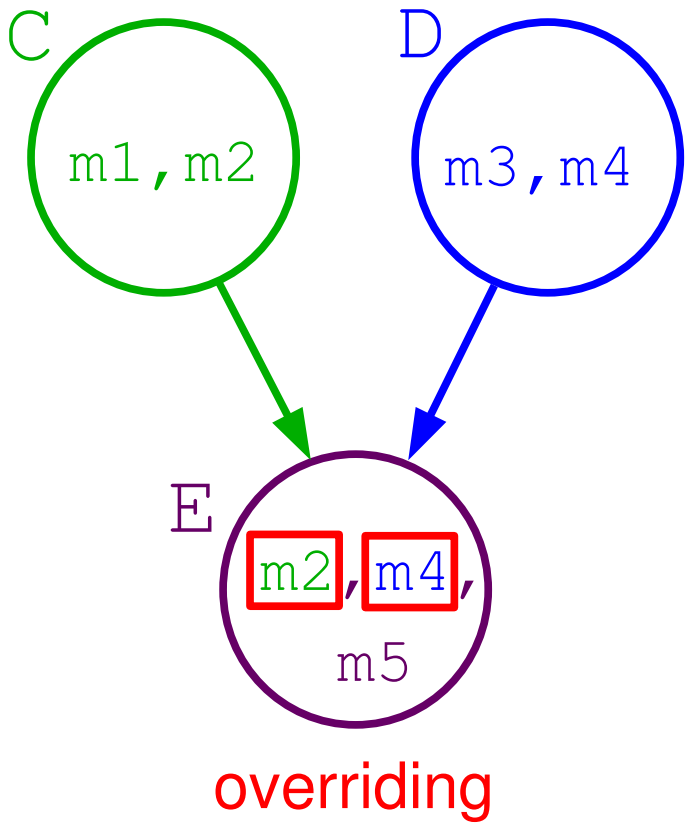


Problem Mehrdeutigkeit:

Welche Methode `m1` (aus `C` oder `D`) soll in `E` verwendet werden?

→ Auflösung muss durch die Programmiersprache definiert werden

Feature 5: Mehrfachvererbung (3)



Dispatch-Tabelle für Klasse `E`:

```
m1_C_C
m2_C_E
m3_D_D
m4_D_E
m5_E_E
```

Feature 5: Mehrfachvererbung (4)

supertyping:

```
convert_E_to_C(e)=e
convert_E_to_D(e)=
    e+sizeof(class C)
```

subtyping:

```
convert_C_to_E(c)=c
convert_D_to_E(d)=
    d-sizeof(class C)
```

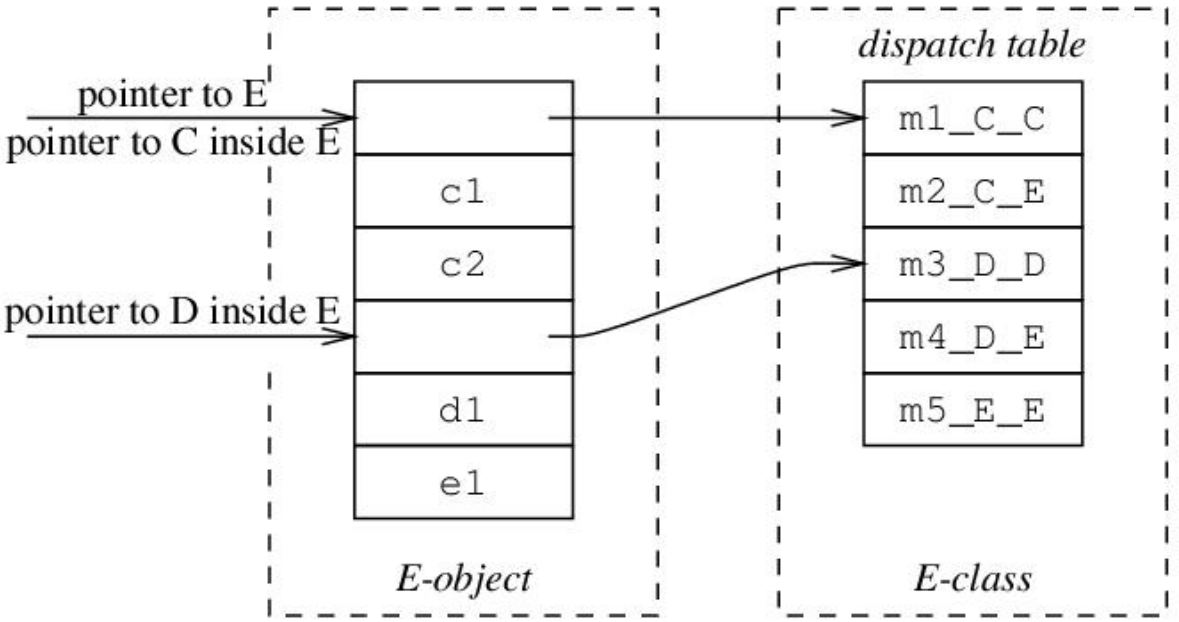


Figure 6.19 A representation of an object of class `E`.

Feature 6: Abhängige Mehrfachvererbung (1)

```
class A {
    field a1;
    field a2;
    method m1();
    method m3();
};

class C extends A {
    field c1;
    field c2;
    method m1();
    method m2();
};

class D extends A {
    field d1;
    method m3();
    method m4();
};

class E extends C, D {
    field e1;
    method m2();
    method m4();
    method m5();
};
```

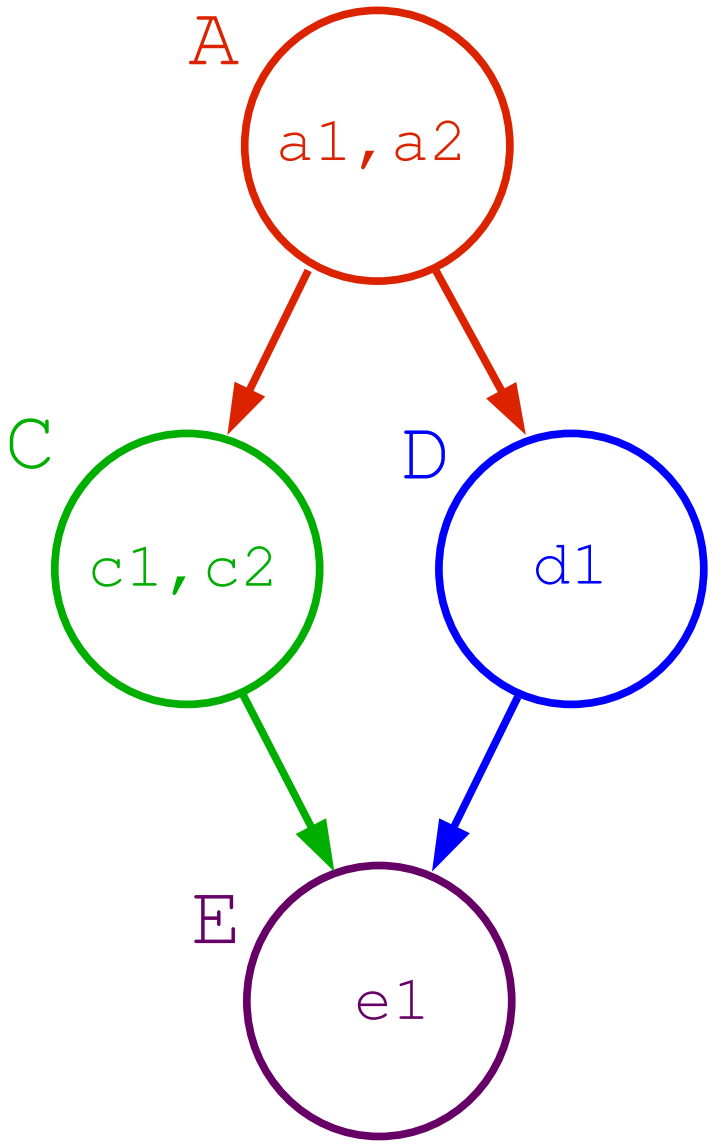
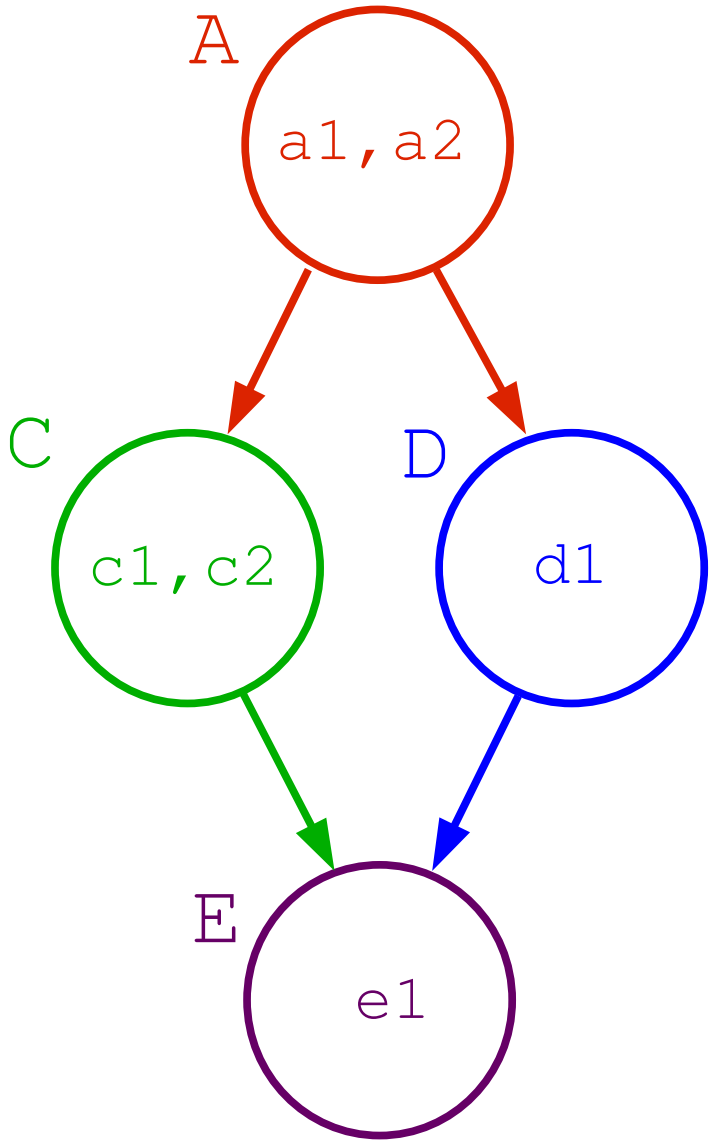


Figure 6.20 An example of dependent multiple inheritance.

Feature 6: Abhängige Mehrfachvererbung (2)



Normalfall:

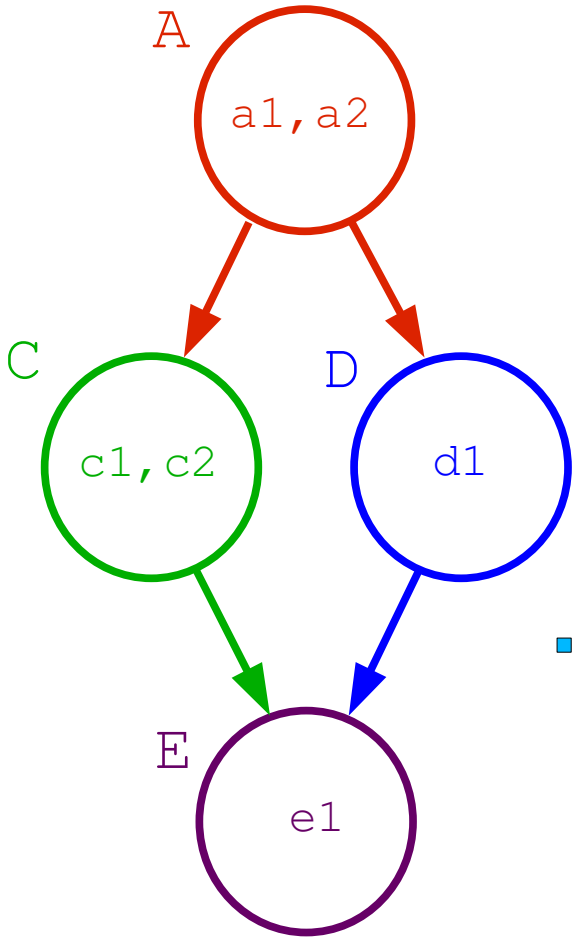
`a1,a2` in `E` nur einmal vorhanden

Problem, Kombination von ...

- alle Felder für `E` in *einem* Deskriptor
- `D`-Felder in `D` hinter `A`-Feldern
- `D`-Felder in `E` nicht hinter `A`-Feldern
  (dort sind schon die `C`-Felder)
- dynamische Bindung

Lösung: Laufzeit-Deskriptor

Feature 6: Abhängige Mehrfachvererbung (3)



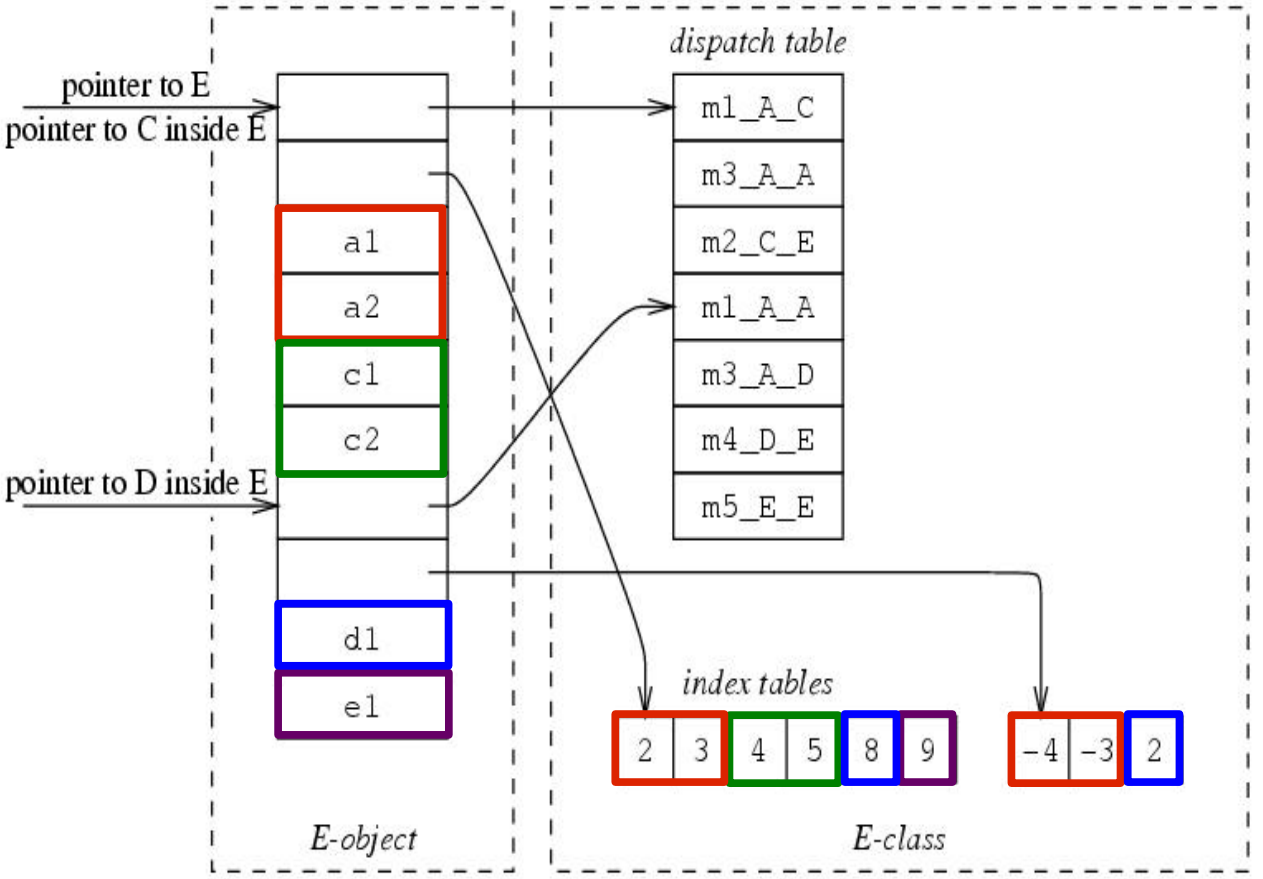


Figure 6.21 An object of class E, with dependent inheritance.

Routinen, Arten

(a) Subroutine (häufigste Form)

- aktiviert, arbeitet, beendet sich
- vergisst Zustand
- gibt Kontrolle an rufende Routine zurück

(b) Iterator

- arbeitet einen Schritt ab, merkt sich Position
- gibt Kontrolle an rufende Routine zurück
- Bsp.: `getchar()`

(c) Koroutine

- setzt Berechnung ab einer gemerkten Position fort
- arbeitet, bis ein bestimmtes Ereignis eintritt
- gibt Kontrolle an eine andere Koroutine ab
- Bsp.: Simulation in Simula, Producer/Consumer-Prinzip

Subroutinen, Aktivierungssegment

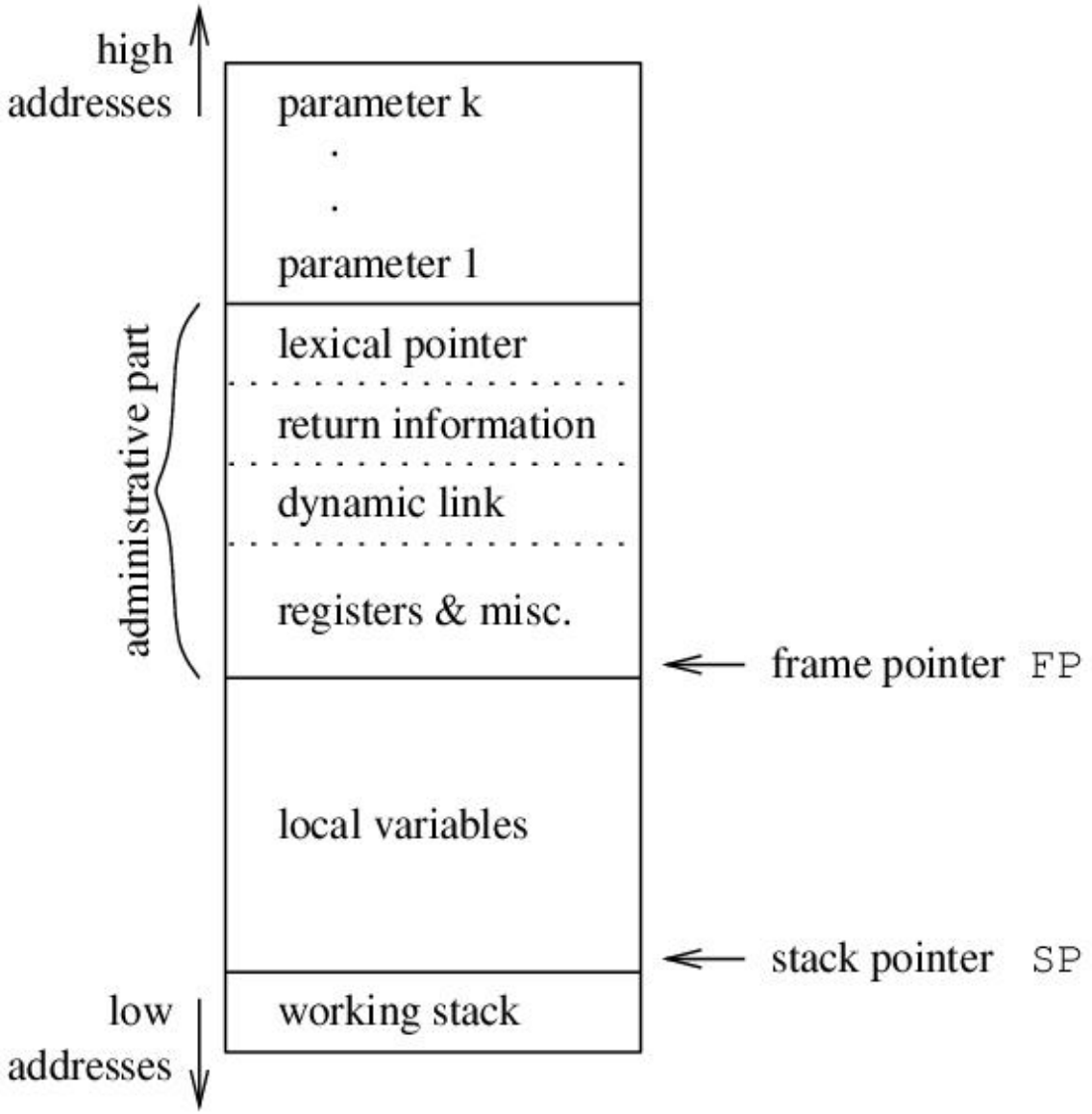
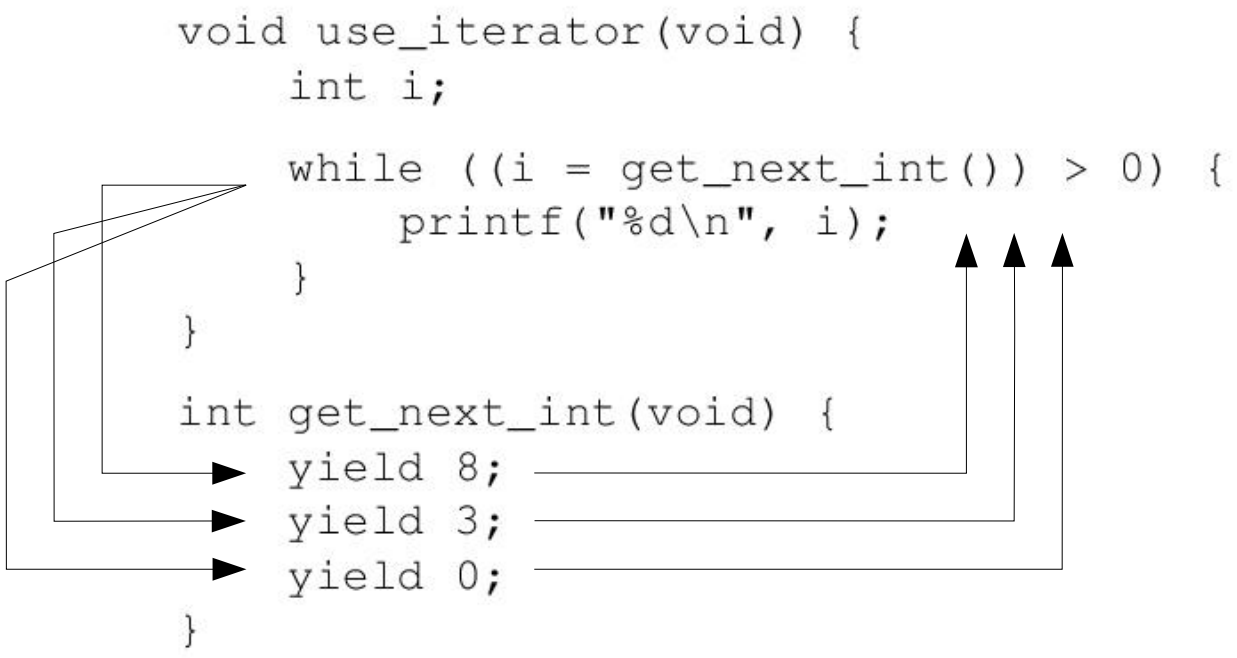


Figure 6.22 Possible structure of an activation record.

Iterator

- arbeitet einen Schritt ab, merkt sich Position
- gibt Kontrolle an rufende Routine zurück (`yield`)



```
void use_iterator(void) {
    int i;

    while ((i = get_next_int()) > 0) {
        printf("%d\n", i);
    }
}

int get_next_int(void) {
    yield 8;
    yield 3;
    yield 0;
}
```

Figure 6.23 An example application of an iterator in C notation.

Koroutinen

- setzt Berechnung ab einer gemerkten Position fort
- arbeitet, bis ein bestimmtes Ereignis eintritt (`empty_buffer`)
- gibt Kontrolle an eine andere Koroutine ab (`resume`)

```
char buffer[100];

void Producer(void) {
    while (produce_buffer()) {
        resume(Consumer);
    }
    empty_buffer();      /* signal end of stream */
    resume(Consumer);
}

void Consumer(void) {
    resume(Producer);
    while (!empty_buffer_received()) {
        consume_buffer();
        resume(Producer);
    }
}
```

Figure 6.24 Simplified producer/consumer communication using coroutines.