Oppgave 7.2

Draw a possible organization for the runtime environment of the following C program, similar to that of Figure 7.4 (page 354).

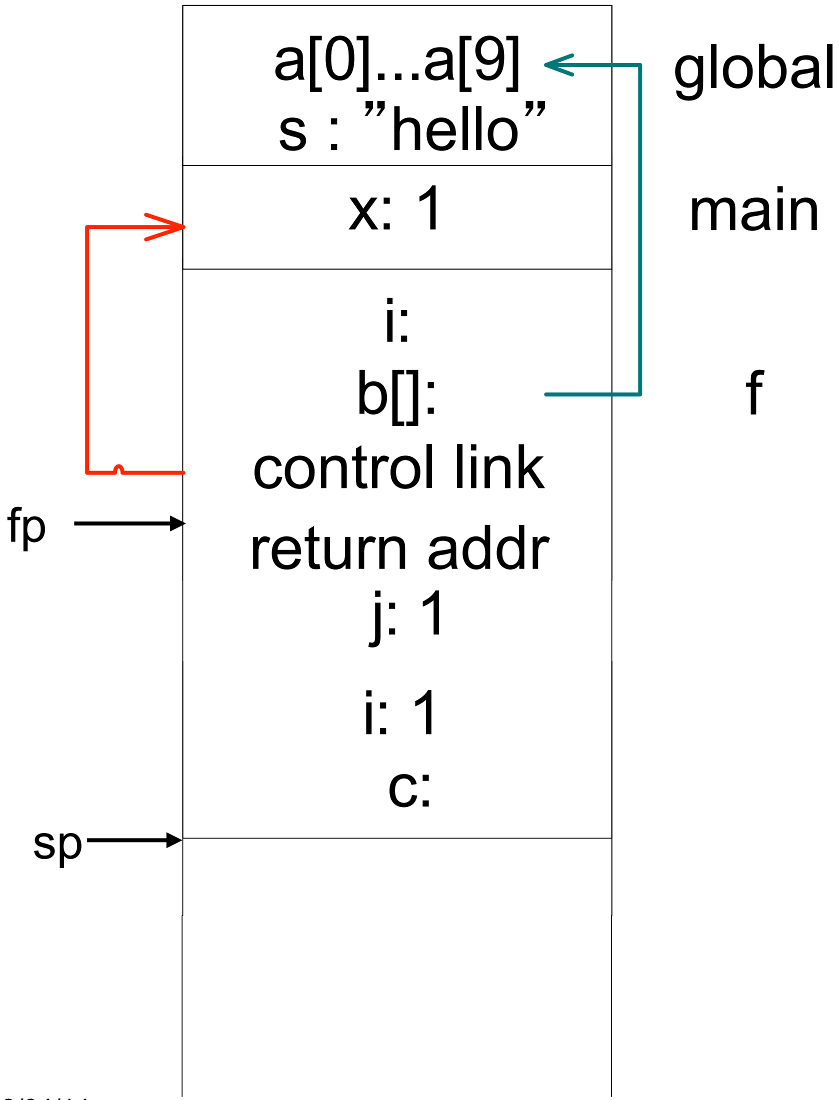
a. After entry into block **A** in function **f**.

b. After entry into block **B** in function **g**.

```
int a[10];
char * s = "hello";

int f(int i, int b[])
{ int j=i;
  A:{ int i=j;
      char c = b[i];
      ...
    }
  return 0;
}
void g(char * s)
{ char c = s[0];
  B:{ int a[5];
      ...
    }
}

main()
{ int x=1;
  x = f(x,a);
  g(s);
  return 0;
}
```

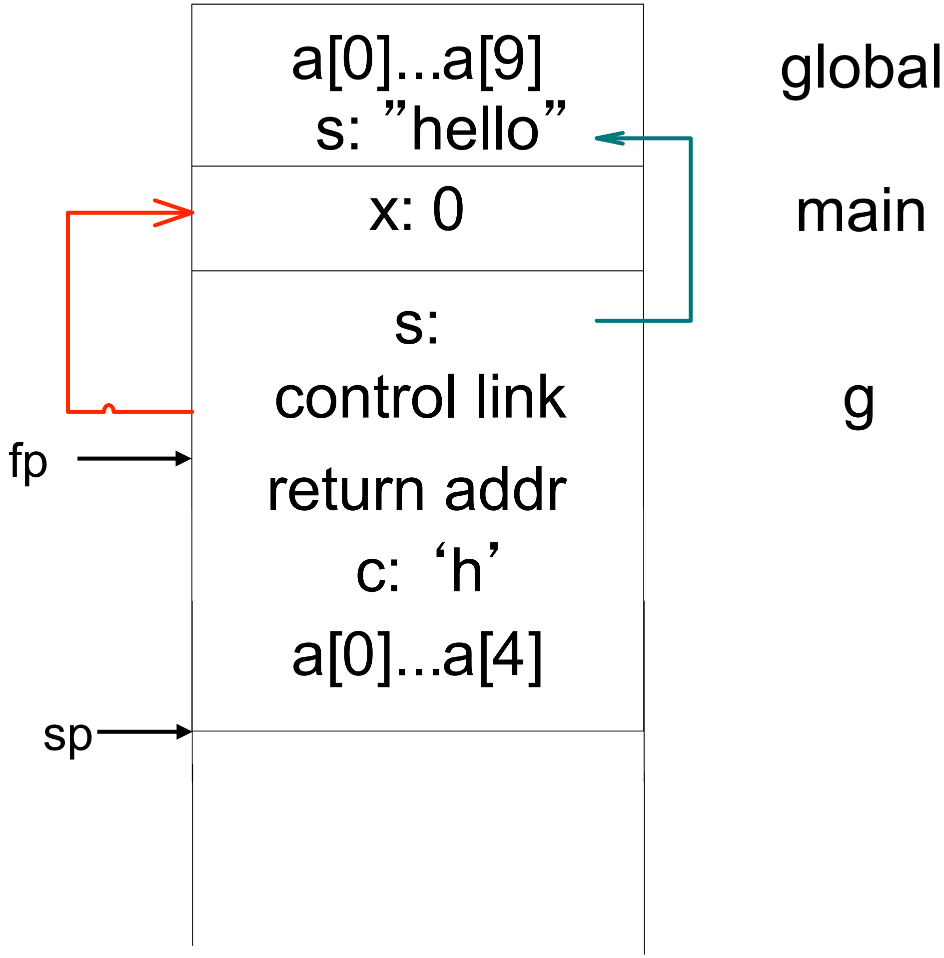
7.2

a.



a[0]...a[9]
s : "hello"
global
x: 1
main
i:
b[]:
f
control link
fp
return addr
j: 1
i: 1
c:
sp

b.



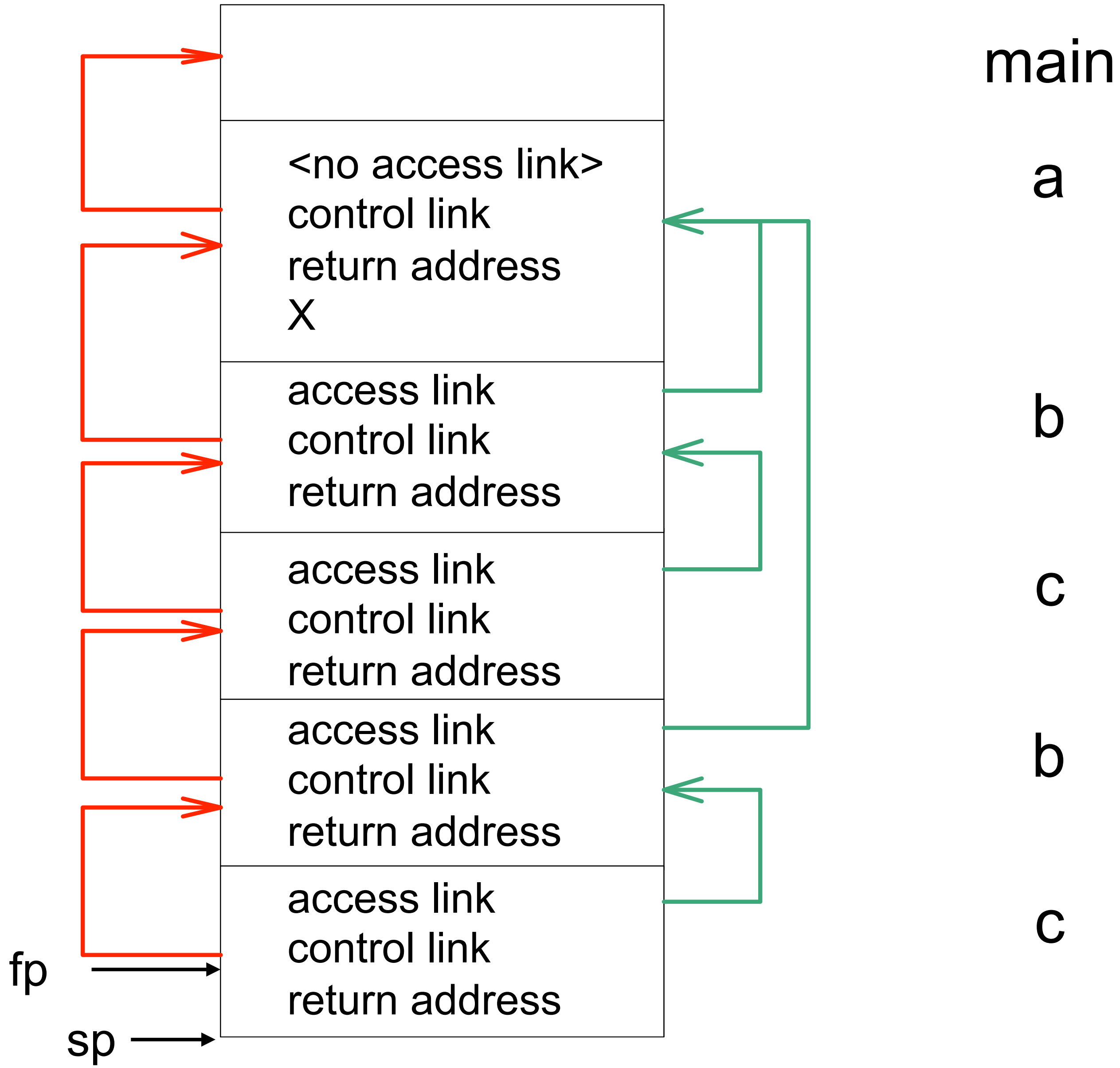
a[0]...a[9]
s: "hello"
global
x: 0
main
s:
control link
g
fp
return addr
c: 'h'
a[0]...a[4]
sp

Oppgave 7.4

Draw the stack of activation records for the following Pascal program, showing the control and access links, after the second call to procedure **c**. Describe how the variable **x** is accessed from within **c**.

```
program env;

procedure a;
var x: integer;

  procedure b;
    procedure c;
    begin
      x := 2;
      b;
    end;
  begin (* b *)
      c;
    end;

begin (* a *)
  b;
end;

begin (* main *)
  a;
end.
```

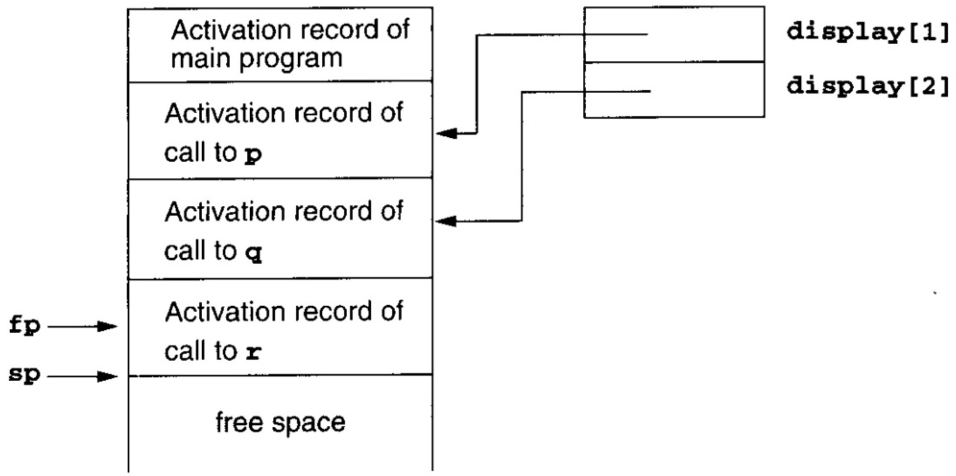
7.4



<no access link>
control link
return address
X
access link
control link
return address
access link
control link
return address
access link
control link
return address
access link
control link
return address
fp
sp
main
a
b
c
b
c

Oppgave 7.10

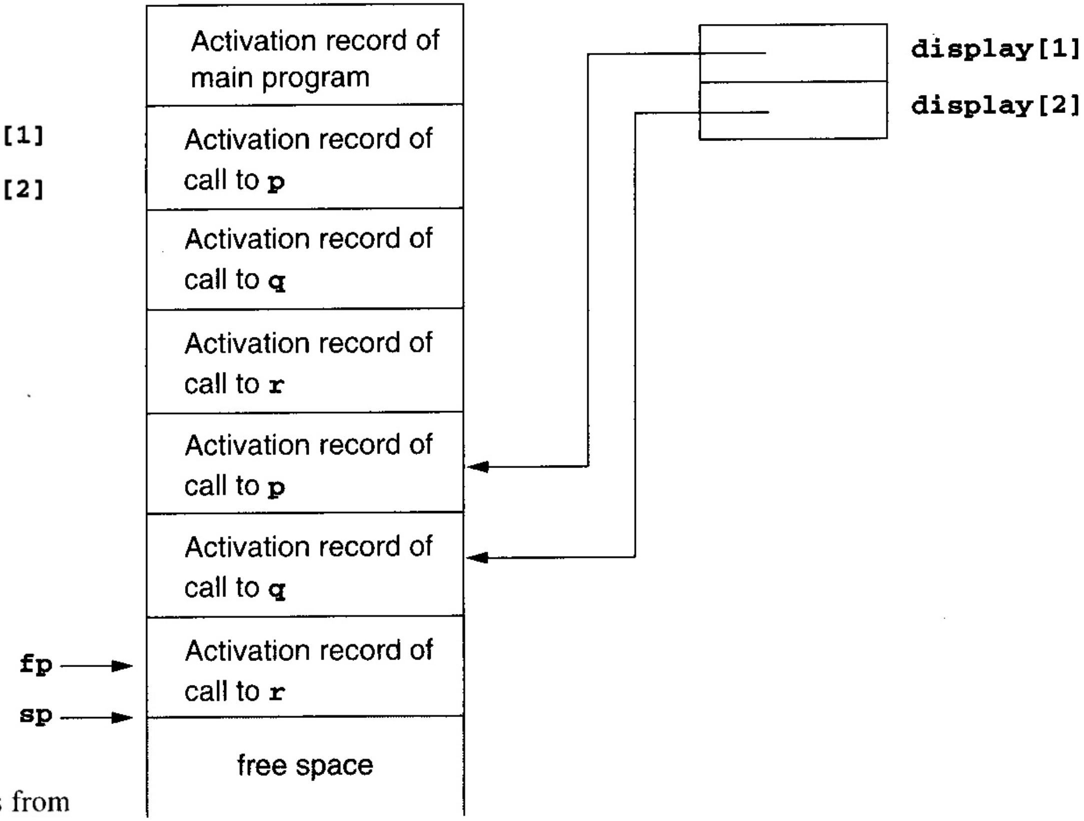
An alternative to access chaining in a language with local procedures is to keep access links in an array outside the stack, indexed by nesting level. This array is called the **display**. For example, the runtime stack of Figure 7.12 (page 369) would look as follows with a display



a. Describe how a display can improve the efficiency of nonlocal references from deeply nested procedures.

b. Redo Exercise 7.4 using a display.

while the runtime stack of Figure 7.13 (page 370) would look as follows:



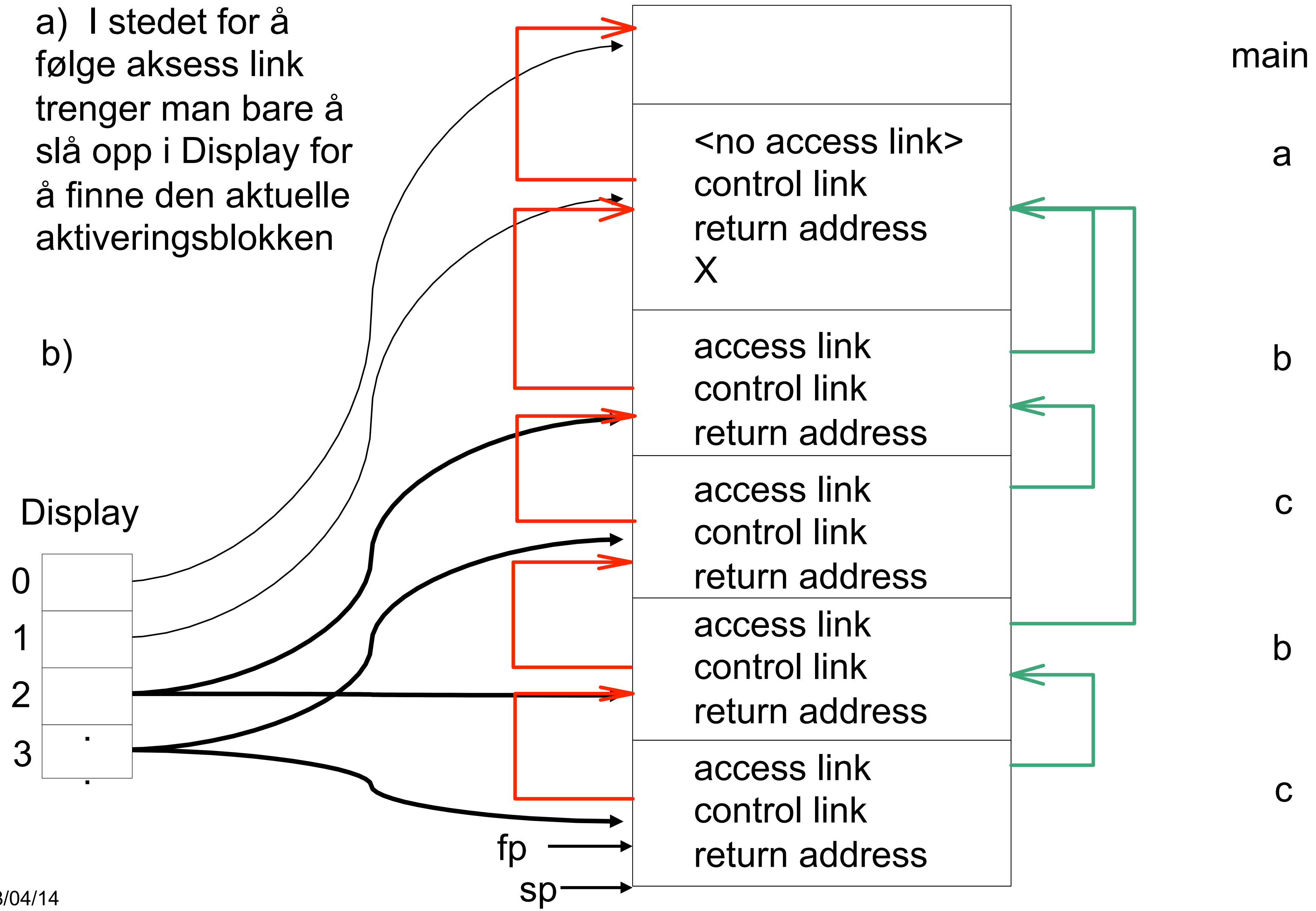
Oppgave 7.4

```
0   program env;

1   procedure a;
    var x: integer;

2     procedure b;
3       procedure c;
        begin
          x := 2;
          b;
        end;
      begin (* b *)
          c;
        end;

    begin (* a *)
      b;
    end;

    begin (* main *)
      a;
    end.
```

7.10

- a) I stedet for å følge aksess link trenger man bare å slå opp i Display for å finne den aktuelle aktiveringsblokken

b)

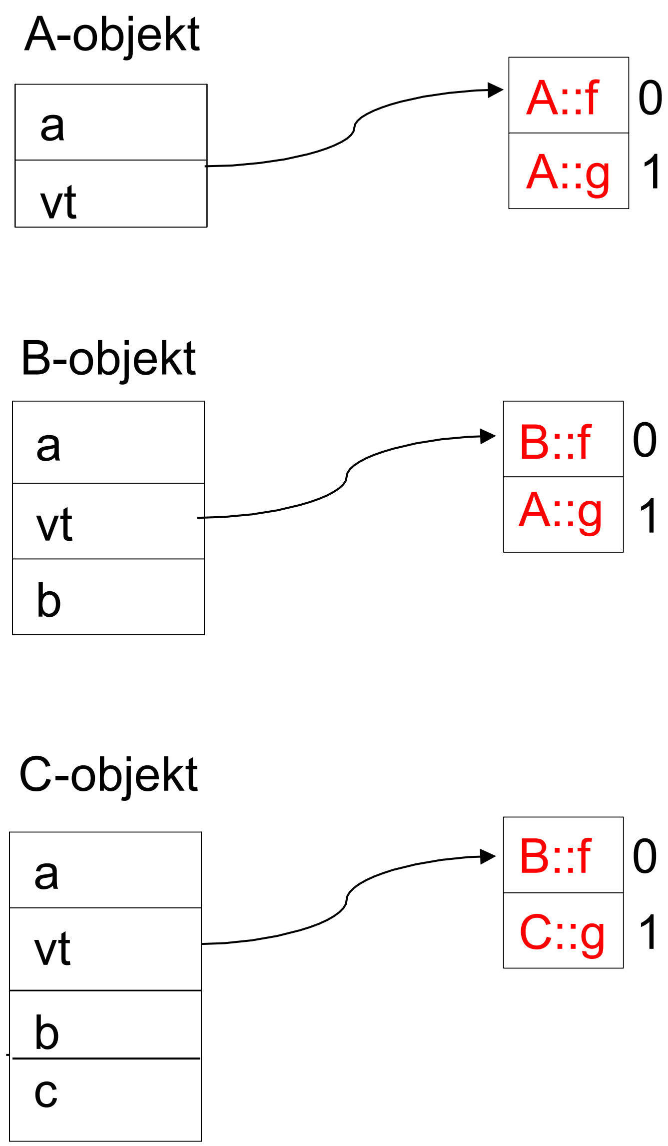


7.13 Draw the memory layout of objects of the following C++ classes, together with the virtual function tables as described in Section 7.4.2:

```
class A
{ public:
  int a;
  virtual void f();
  virtual void g();
};
class B : public A
{ public:
  int b;
  virtual void f();
  void h();
};
class C : public B
{ public:
  int c;
  virtual void g();
}
```

7.13 Draw the memory layout of objects of the following C++ classes, together with the virtual function tables as described in Section 7.4.2:

```
class A
{ public:
  int a;
  virtual void f();
  virtual void g();
};
class B : public A
{ public:
  int b;
  virtual void f();
  void h();
};
class C : public B
{ public:
  int c;
  virtual void g();
}
```

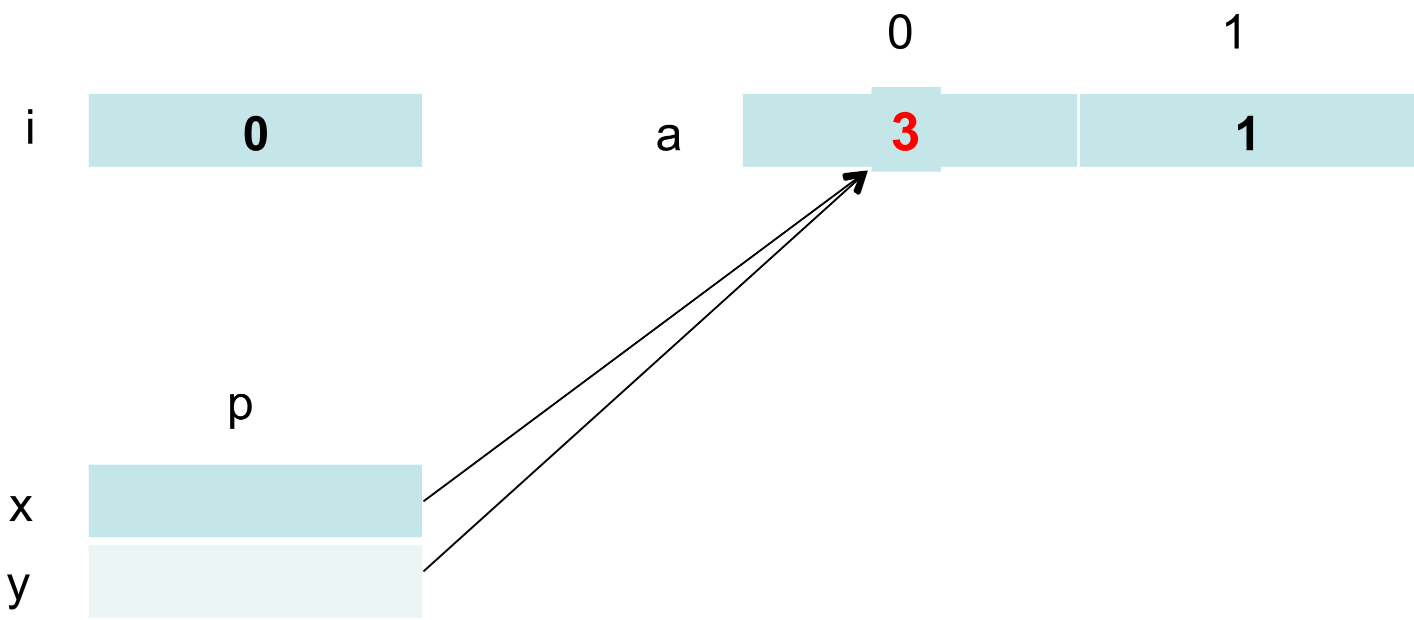


7.15 Give the output of the following program (written in C syntax) using the four parameter passing methods discussed in Section 7.5:

```
#include <stdio.h>
int i=0;

void p(int x, int y)
{ x += 1;
  i += 1;
  y += 1;
}

main()
{ int a[2]={1,1};
  p(a[i],a[i]);
  printf("%d %d\n",a[0],a[1]);
  return 0;
}
```

by value	by reference
1 1	
by value-result	**by name**

7.15 by reference



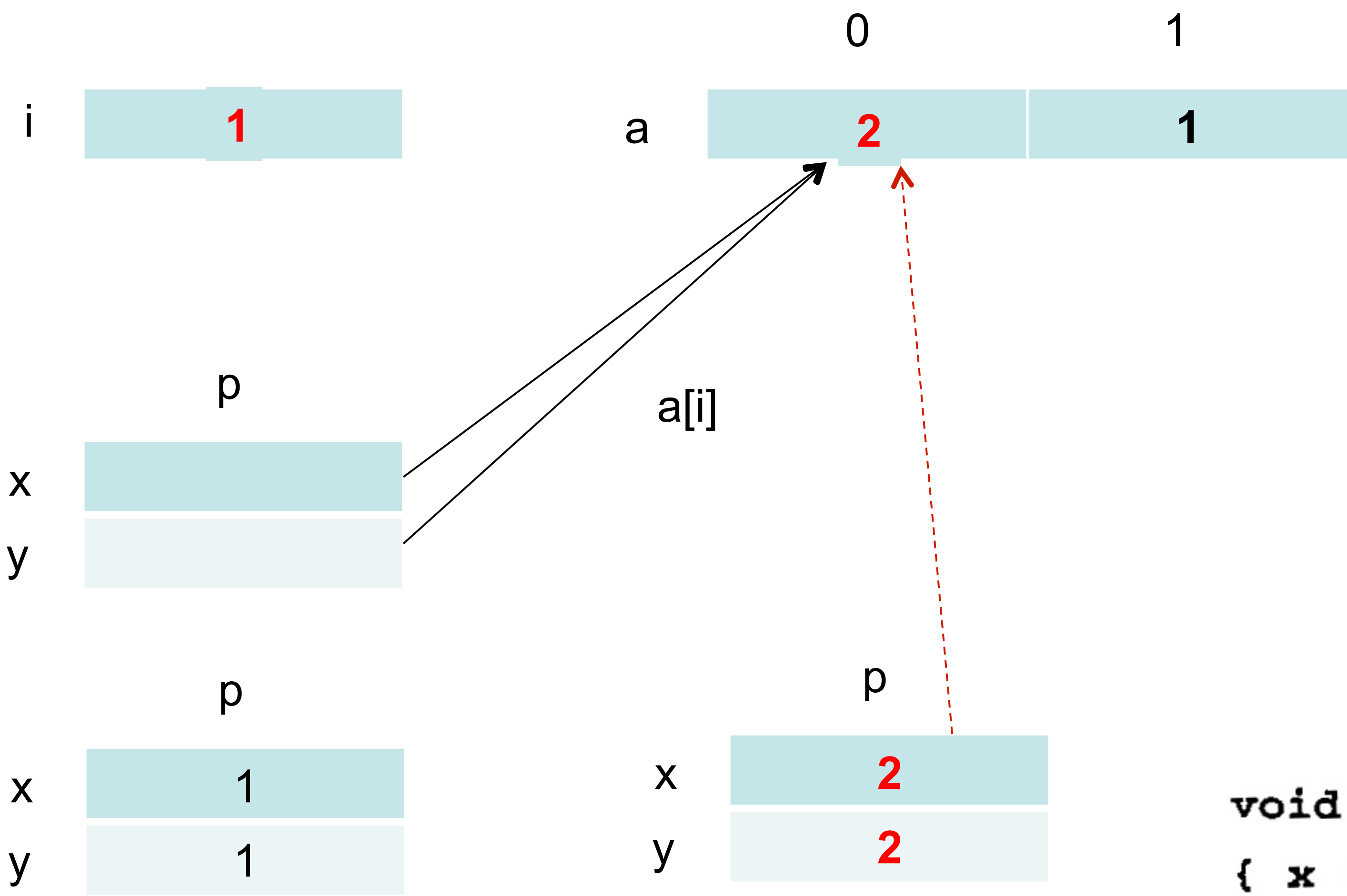

```
void p(int x, int y)
{ x += 1;
  i += 1;
  y += 1;
}
```

7.15 Give the output of the following program (written in C syntax) using the four parameter passing methods discussed in Section 7.5:

```
#include <stdio.h>
int i=0;

void p(int x, int y)
{ x += 1;
  i += 1;
  y += 1;
}

main()
{ int a[2]={1,1};
  p(a[i],a[i]);
  printf("%d %d\n",a[0],a[1]);
  return 0;
}
```

by value	by reference
1 1	3 1
by value-result	**by name**

7.15 by value-result – address at call



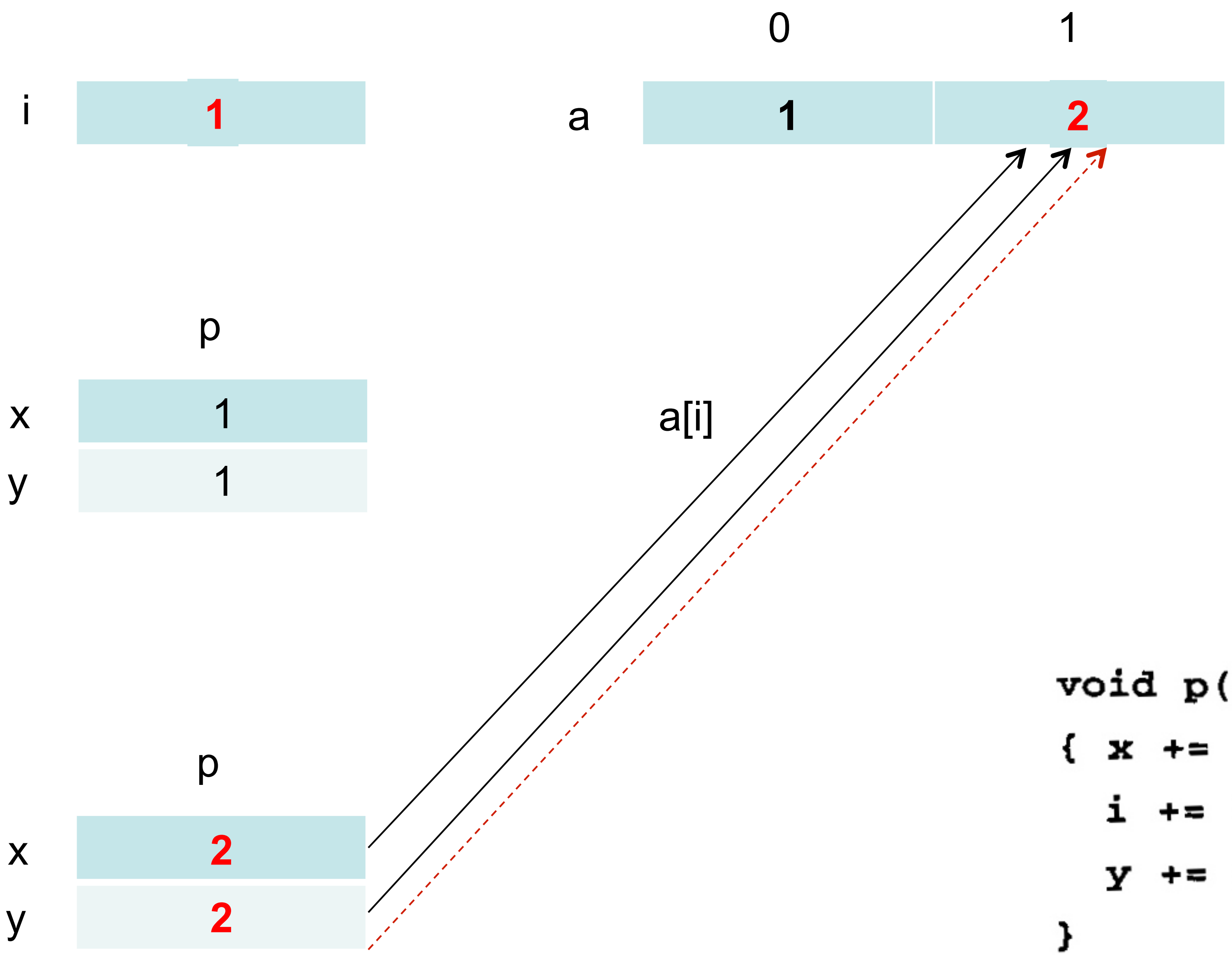

```
void p(int x, int y)
{ x += 1;
  i += 1;
  y += 1;
}
```

7.15 Give the output of the following program (written in C syntax) using the four parameter passing methods discussed in Section 7.5:

```
#include <stdio.h>
int i=0;

void p(int x, int y)
{ x += 1;
  i += 1;
  y += 1;
}

main()
{ int a[2]={1,1};
  p(a[i],a[i]);
  printf("%d %d\n",a[0],a[1]);
  return 0;
}
```

by value	by reference
1 1	3 1
by value-result	**by name**
2 1	

7.15 by value-result – address at exit




```
void p(int x, int y)
{ x += 1;
  i += 1;
  y += 1;
}
```

7.15 Give the output of the following program (written in C syntax) using the four parameter passing methods discussed in Section 7.5:

```
#include <stdio.h>
int i=0;

void p(int x, int y)
{ x += 1;
  i += 1;
  y += 1;
}

main()
{ int a[2]={1,1};
  p(a[i],a[i]);
  printf("%d %d\n",a[0],a[1]);
  return 0;
}
```

by value	by reference
1 1	3 1
by value-result	**by name**
2 1 1 2	

7.15 by name

a(i) = a(i) + 1 == a(0) = a(0) + 1 = 1 + 1 = 2

i = i +1 == i = 0 +1 = 1

a(i) = a(i) + 1 == a(1) = a(1) + 1 = 1 + 1 = 2

```
void p(int x, int y)
{ x += 1;
  i += 1;
  y += 1;
}
```

```
p(a[i],a[i]);
```

7.15 Give the output of the following program (written in C syntax) using the four parameter passing methods discussed in Section 7.5:

```
#include <stdio.h>
int i=0;

void p(int x, int y)
{ x += 1;
  i += 1;
  y += 1;
}

main()
{ int a[2]={1,1};
  p(a[i],a[i]);
  printf("%d %d\n",a[0],a[1]);
  return 0;
}
```

by value	by reference
1 1	3 1
by value-result	**by name**
2 1 1 2	2 2

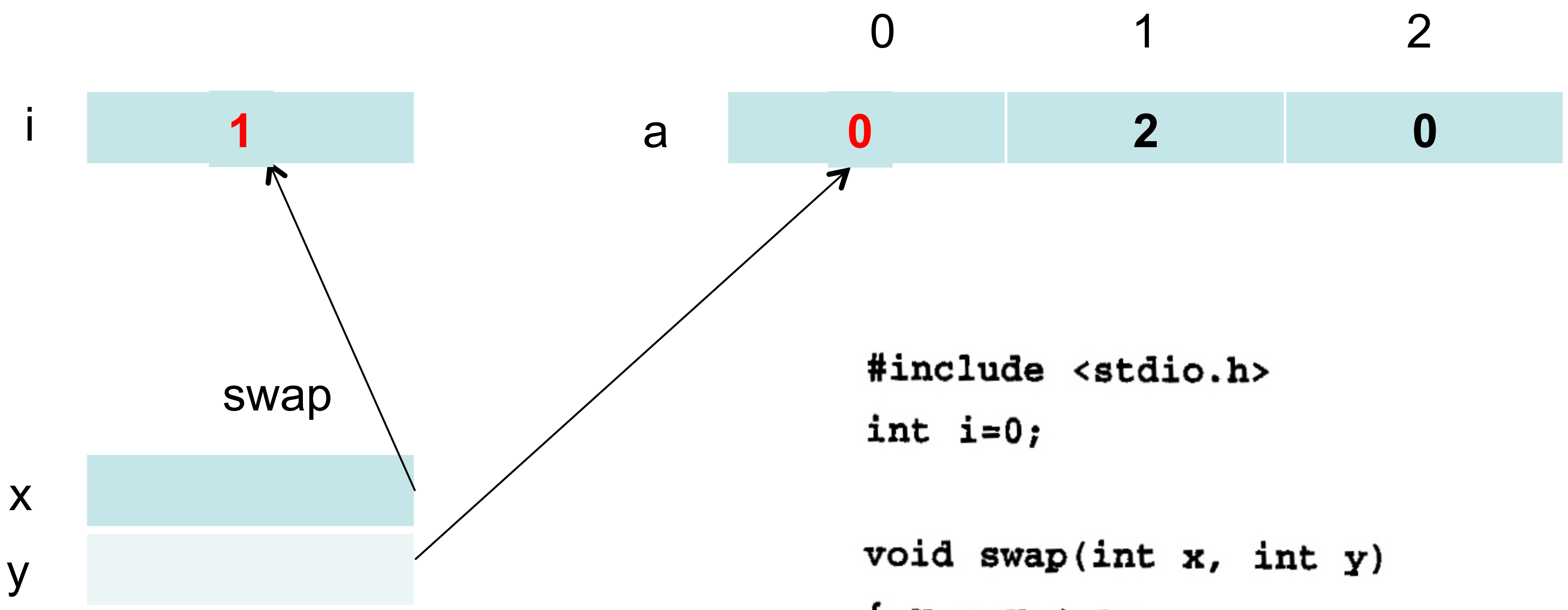
7.16

Give the output of the following program (in C syntax) using the four parameter passing methods of Section 7.5:

```
#include <stdio.h>
int i=0;

void swap(int x, int y)
{ x = x + y;
  y = x - y;
  x = x - y;
}

main()
{ int a[3] = {1,2,0};
   swap(i,a[i]);
   printf("%d %d %d %d\n",i,a[0],a[1],a[2]);
   return 0;
}
```

by value	by reference
0 1 2 0	
by value-result	**by name**

7.16 by reference




```
#include <stdio.h>
int i=0;

void swap(int x, int y)
{ x = x + y;
  y = x - y;
  x = x - y;
}

main()
{ int a[3] = {1,2,0};
   swap(i,a[i]);
   printf("%d %d %d %d\n",i,a[0],a[1],a[2]);
   return 0;
}
```

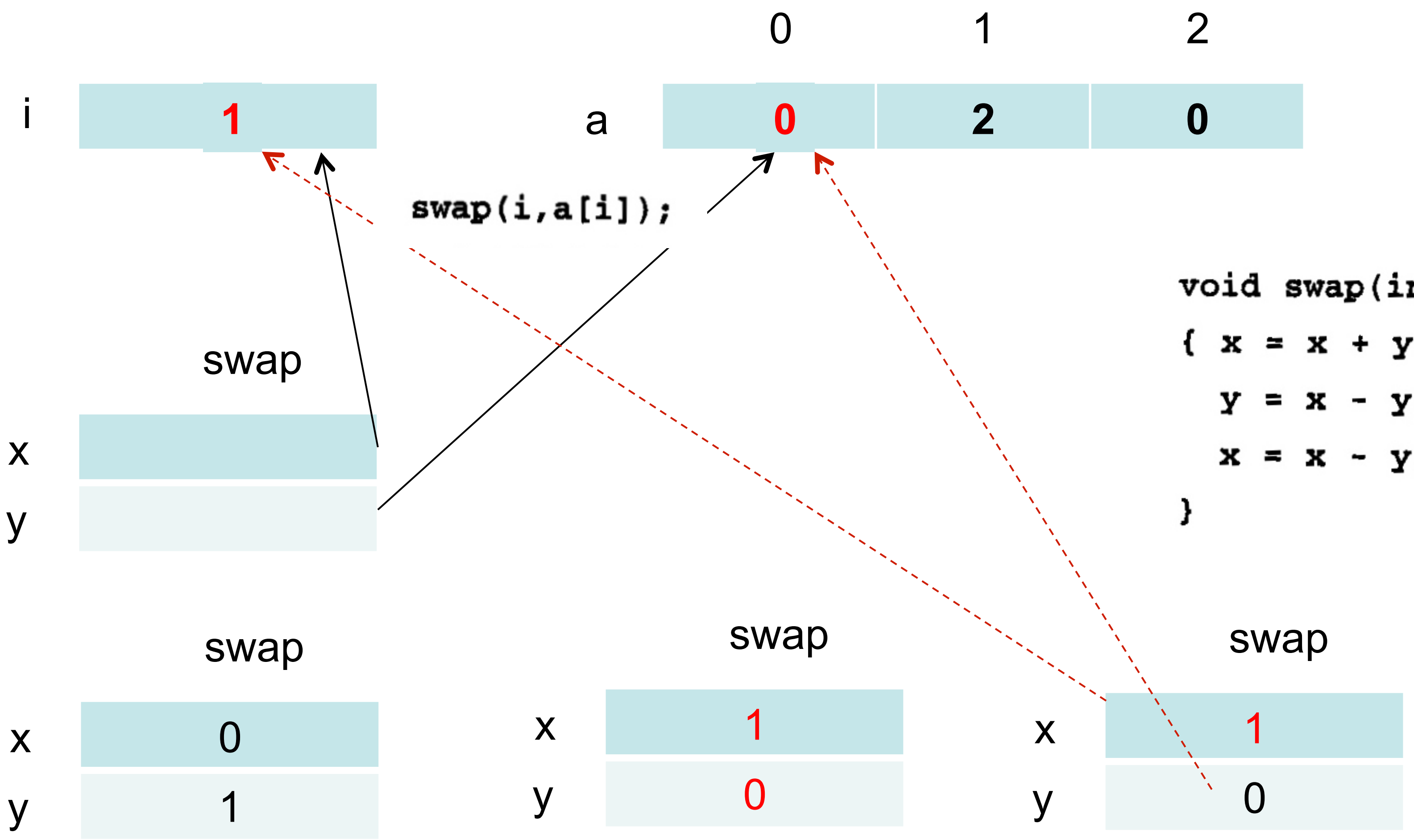
7.16

Give the output of the following program (in C syntax) using the four parameter passing methods of Section 7.5:

```
#include <stdio.h>
int i=0;

void swap(int x, int y)
{ x = x + y;
  y = x - y;
  x = x - y;
}

main()
{ int a[3] = {1,2,0};
   swap(i,a[i]);
   printf("%d %d %d %d\n",i,a[0],a[1],a[2]);
   return 0;
}
```

by value	by reference
0 1 2 0	1 0 2 0
by value-result	**by name**

7.16 by value-result – address at call




```
void swap(int x, int y)
{ x = x + y;
  y = x - y;
  x = x - y;
}
```

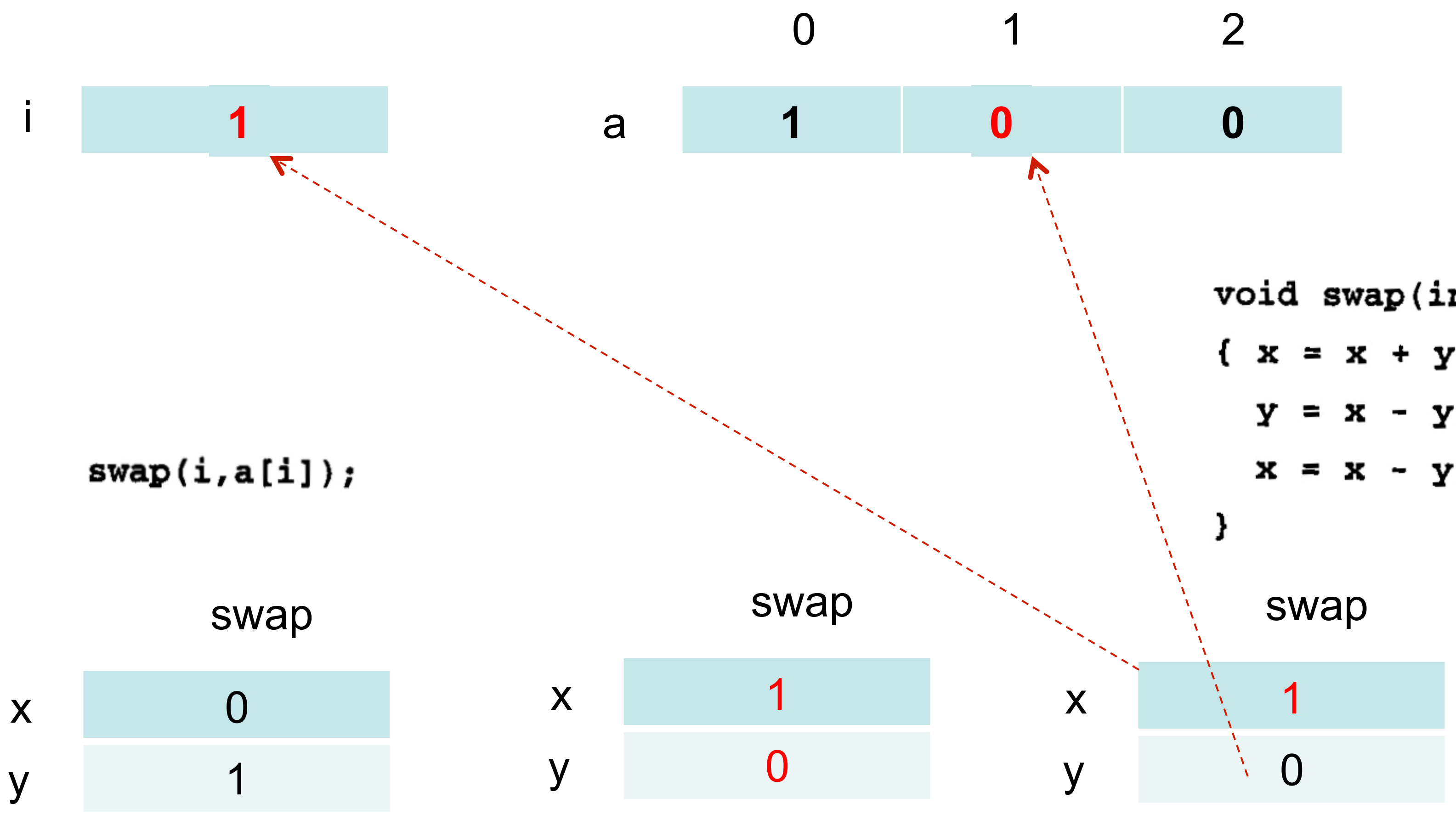
7.16

Give the output of the following program (in C syntax) using the four parameter passing methods of Section 7.5:

```
#include <stdio.h>
int i=0;

void swap(int x, int y)
{ x = x + y;
  y = x - y;
  x = x - y;
}

main()
{ int a[3] = {1,2,0};
   swap(i,a[i]);
   printf("%d %d %d %d\n",i,a[0],a[1],a[2]);
   return 0;
}
```

by value	by reference
0 1 2 0	1 0 2 0
by value-result	**by name**
1 0 2 0	

7.16 by value-result – address at exit




```
swap(i,a[i]);
```

```
void swap(int x, int y)
{ x = x + y;
  y = x - y;
  x = x - y;
}
```

7.16

Give the output of the following program (in C syntax) using the four parameter passing methods of Section 7.5:

```
#include <stdio.h>
int i=0;

void swap(int x, int y)
{ x = x + y;
  y = x - y;
  x = x - y;
}

main()
{ int a[3] = {1,2,0};
    swap(i,a[i]);
    printf("%d %d %d %d\n",i,a[0],a[1],a[2]);
    return 0;
}
```

by value	by reference
0 1 2 0	1 0 2 0
by value-result	**by name**
1 0 2 0 1 1 0 0	

7.16 by name

i = i + a(i)	==	i= 0 + 1 = 1		
a(i) = i - a(i)	==	a(1) = 1 – a(1)	==	a(1) = 1 – 2 = -1
i = i - a(i)	==	i = 1 – a(1)	==	i = 1 – (-1) = 2

```
swap(i,a[i]);
```

```
void swap(int x, int y)
{ x = x + y;
  y = x - y;
  x = x - y;
}
```

7.16

Give the output of the following program (in C syntax) using the four parameter passing methods of Section 7.5:

```
#include <stdio.h>
int i=0;

void swap(int x, int y)
{ x = x + y;
  y = x - y;
  x = x - y;
}

main()
{ int a[3] = {1,2,0};
   swap(i,a[i]);
   printf("%d %d %d\n",i,a[0],a[1],a[2]);
   return 0;
}
```

by value	by reference
0 1 2 0	1 0 2 0
by value-result	**by name**
1 0 2 0 1 1 0 0	2 1 -1 0

Eksamen 2005 a)

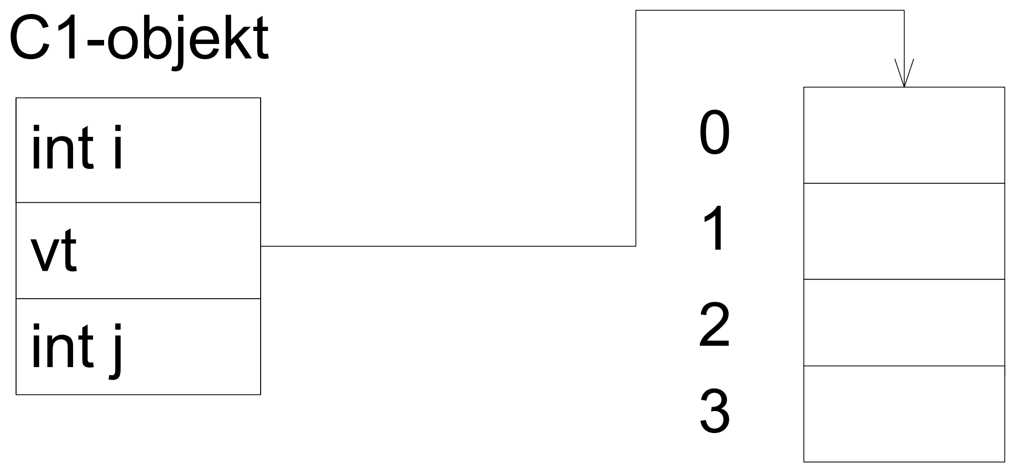
Anta at vi har et språk med klasser og subklasser. Alle metoder er virtuelle, slik at de kan redefineres i subklasser.
Gitt følgende klassedefinisjoner:

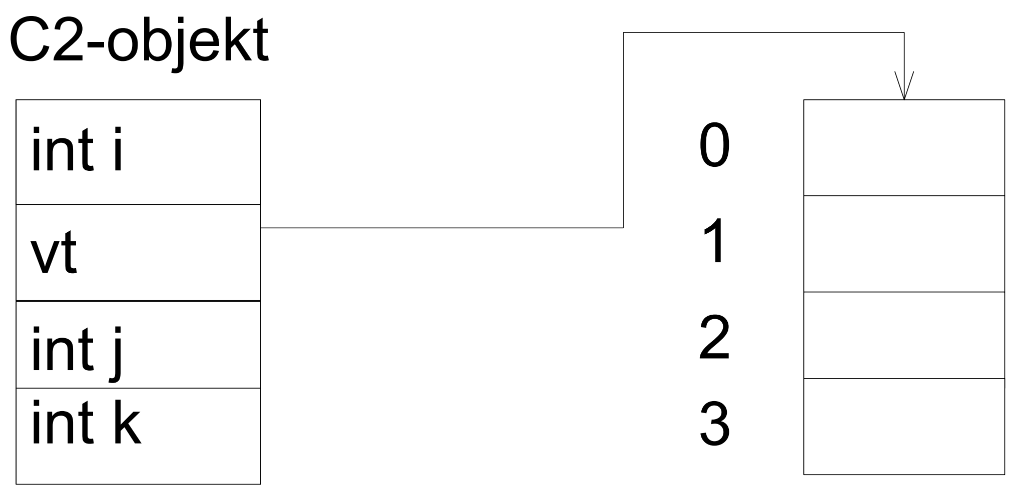
```
class A { int i;
          void P {... AP ...};
          void Q {... AQ ...};   }
class B extends A { int j;
                    void Q {... BQ ...};
                    void R {... BR ...};   }
class C1 extends B {
  void P {... C1P ...};
  void S {... C1S ...};   }
class C2 extends B { int k;
                     void R {... C2R ...};
                     void T {... C2T ...};   }
```

Vis hvordan objekter av klassene C1 og C2 vil være strukturert (layout) og tegn virtuell-tabellen for hvert av objektene. Bruk navnene i metodekroppene til å angi hvilken definisjon som gjelder for hvert objekt.

Eksamen 2005

a)





Eksamen 2005

b)

Vi finner nå på å innføre i språket muligheten for å spesifisere en metode til å være **final**. Det skal bety at den ikke lenger er virtuell, dvs at den ikke kan redefineres i subklasser.

Anta at vi i klassen B spesifiserer metoden Q til å være **final**.

Må vi da endre på virtuell-tabellen for B-objekter?

Begrund svaret.

NEI, virtuelle metoder i B-objekter kan fremdeles kaldes via A-typede pekere.

Eksamen 2005 c)

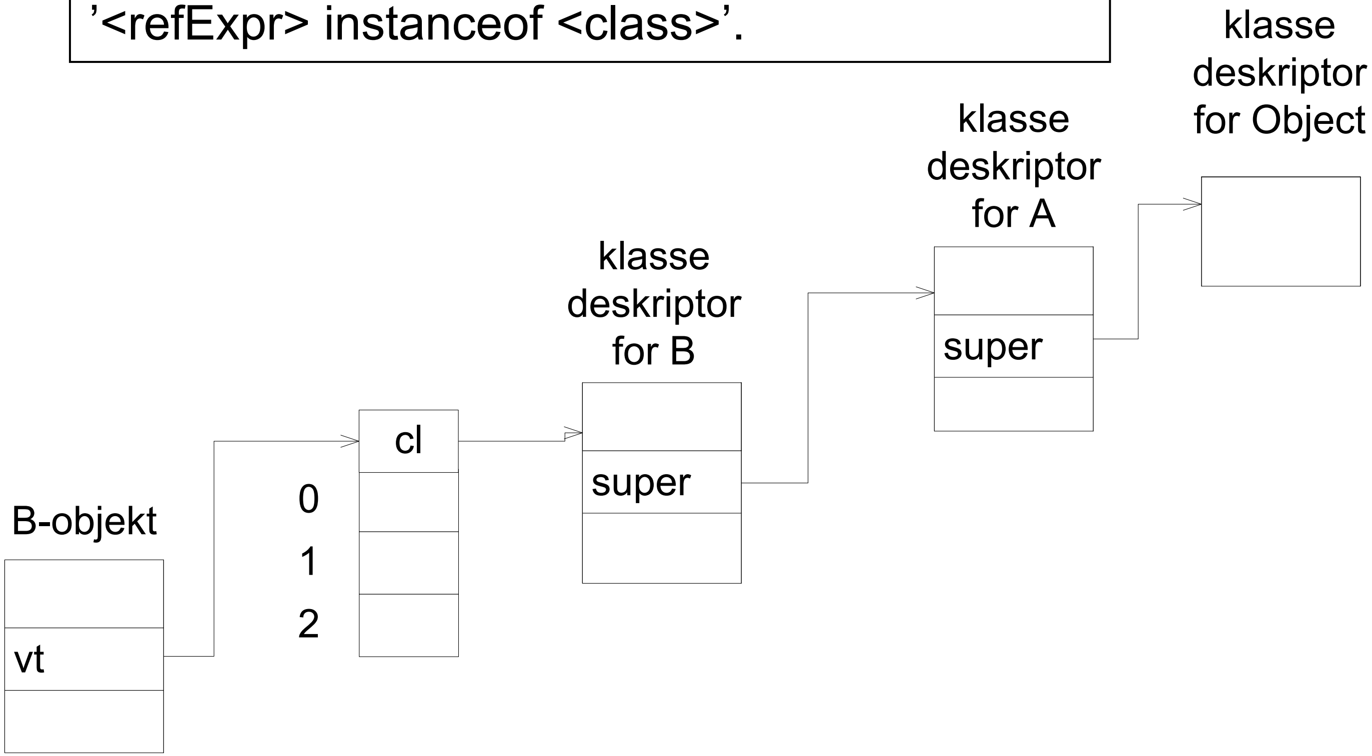
Vi innfører nå operatoren 'instanceof': Uttrykket

'<refExpr> instanceof <class>'

er True hvis objektet som <refExpr> peker på har en klasse som er klassen <class> eller en subklasse av klassen <class>, ellers False.

For å kunne implementere denne operatoren utvider vi virtuell-tabellen med en peker til klasse-deskriptoren, som det er en av for hver klasse i programmet. Klassedeskriptoren har en variabel 'super', som peker til klasse-deskriptoren for superklassen. Klasser uten eksplisitt superklasse har den spesielle klasse Object som super. Eksemplet viser dette for et objekt av klassen B.

Skisser algoritmen som beregner verdien av '<refExpr> instanceof <class>'.



Eksamen 2005

c)

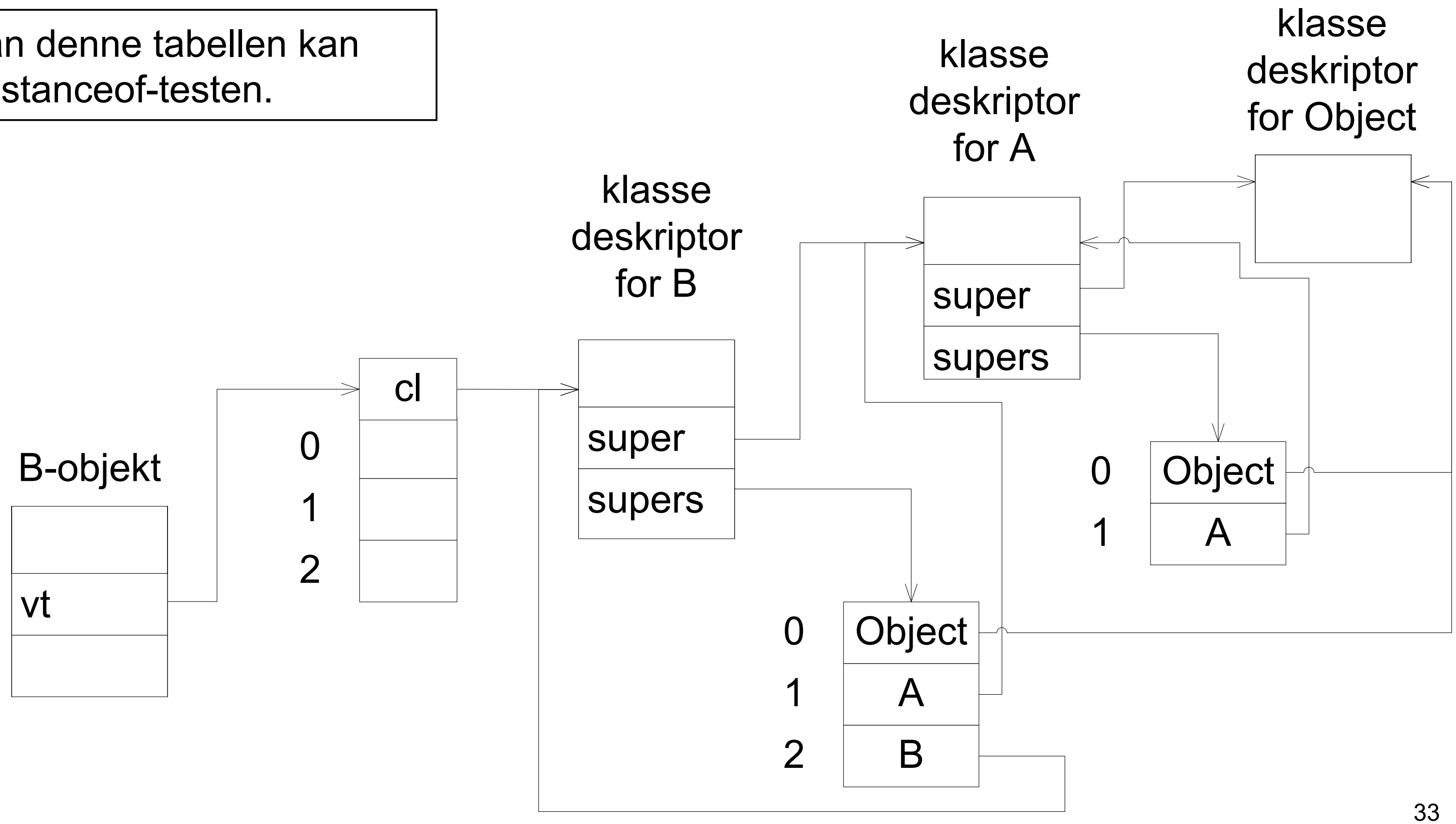
’<refExpr> instanceof <class>’

```
instanceof = false;
cd = <refExpr>.vt.cl;
while not(cd = klassedeskriptor for klassen Object) do
  { if cd = klassedeskriptor for klassen <class>
    then instanceof = true;
    cd = cd.super
  }
```

Eksamen 2005 d)

For å effektivisere testen på 'instanceof' innfører vi at en klassedeskriptor har en tabell 'supers', som inneholder superklassene for klassen samt klassen selv. Denne tabellen har som indeks klassens 'subklassenivå' startende med 0 for Object, 1 for rotklassen i et subklassehierarki, 2 for neste nivå, osv. I vårt eksempel har klassen A subklassenivå 1, B har 2, C1 og C2 har begge 3.

Forklar hvordan denne tabellen kan effektivisere instanceof-testen.



Eksamen 2005 d)

Til illustrere denne forklaringen kan du bruke følgende:

Vi innfører nå ytterligere to klasser:

```
class C11 extends C1 {...}
class C21 extends C2 {...}
```

Lag supers-tabellene for klassedeskriptorene for C11 og C21, og vis hvordan følgende tester gjøres:

```
rC11 = new C11;

rC11 instanceof C1    (1)
rC11 instanceof C2    (2)
```

Eksamen 2005

d) Supers for C11 og C21

0	Object
1	A
2	B
3	C1
4	C11

0	Object
1	A
2	B
3	C2
4	C21

Eksamen 2005

d)

Generell test:

1. Sjekk at <class>.subklassenivå ligger innenfor grensene på den aktuelle supers-tabellen
2. <refExpr>.vt.cl.supers[<class>.subklassenivå] = <class>

Konkrete tester:

```
rc11.vt.cl.supers(3) = C1   dvs C1 = C1    dvs true
rc11.vt.cl.supers(3) = C2   dvs C1 = C2    dvs false
```