F03 - Dynamiskt minne i C

5DV149 Datastrukturer och algoritmer
Programmeringsbok i C

Niclas Börlin

niclas.borlin@cs.umu.se

2024-04-02 Tis

Innehåll

- Repetition lokala variabler, parameteröverföring, pekare
- Dynamiskt minne i C
- Generiska datatyper i C
- Hur man implementerar en länkad lista
- Datatyper i kodbasen — `kill_function` eller inte?

Minne och variabler

- När ett program körs så hamnar olika delar av programmet i olika delar av minnet

Typ av data	Typ av minne
Exekverbar kod	Icke skrivbart minne
Statisk text och konstanter	Icke skrivbart minne
Lokala variabler	Stacken
Dynamiskt allokerat minne	Heapen

- Stacken är ett eget reserverat minne
  - Relativt litet, 8 MB på min maskin
- Heapen är i princip resten av det tillgängliga minnet
  - 32 GB på min maskin

Lokala variabler och parameteröverföring

Kodexempel med funktionsanrop

- Här är ett program med två funktioner, `print()` och `add()`

code/vars-on-stack.c

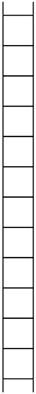
```c
#include <stdio.h>
void print(int v)
{
    printf("v = %d\n",v);
}
int add(int c, int d)
{
    int a;
    a = c + d;
    return a;
}
int main(void)
{
    int a = 2;
    int c = 3;
    int sum;
    sum = add(a, c);
    print(sum);
    sum = add(sum, c + 4);
    print(sum);
    return 0;
}
```

Lokala variabler och stacken (1)

- Lokala variabler lagras i en minnesarea som kallas stacken

code/vars-on-stack.c

```c
int main(void)
{
    int a = 2;
    int c = 3;
    int sum;
```



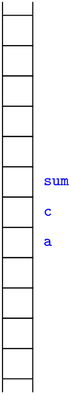
Lokala variabler och stacken (1)

- Lokala variabler lagras i en minnesarea som kallas stacken

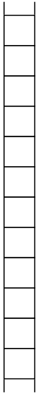
code/vars-on-stack.c

```
int main(void)
{
    int a = 2;
    int c = 3;
    int sum;
```

- När en funktion anropas så reserveras/allokeras minne för variablerna automatiskt



Lokala variabler och stacken (1)



- Lokala variabler lagras i en minnesarea som kallas stacken

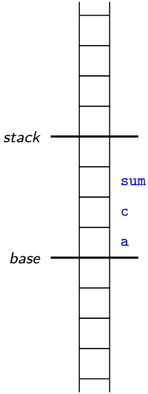
code/vars-on-stack.c

```c
int main(void)
{
    int a = 2;
    int c = 3;
    int sum;
```

- När en funktion anropas så reserveras/allokeras minne för variablerna automatiskt
- Vid återhopp så frigörs/deallokeras minnet automatiskt

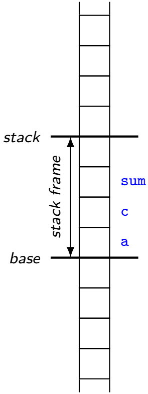
Lokala variabler och stacken (2)

- Internt används två pekare (*base* och *stack*) för att hålla reda på funktionens del av stacken



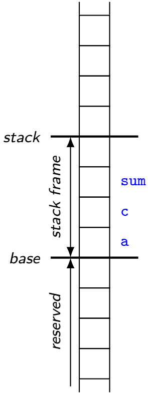
Lokala variabler och stacken (2)

- Internt används två pekare (*base* och *stack*) för att hålla reda på funktionens del av stacken
  - Området mellan pekarna kallas för *stack frame* (aktiveringspost)



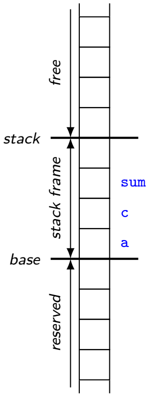
Lokala variabler och stacken (2)

- Internt används två pekare (*base* och *stack*) för att hålla reda på funktionens del av stacken
  - Området mellan pekarna kallas för *stack frame* (aktiveringspost)
  - Området under kan betraktas som upptaget



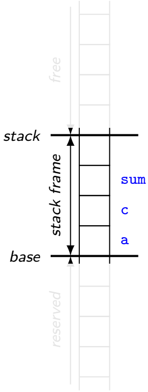
Lokala variabler och stacken (2)

- Internt används två pekare (*base* och *stack*) för att hålla reda på funktionens del av stacken
  - Området mellan pekarna kallas för *stack frame* (aktiveringspost)
  - Området under kan betraktas som upptaget
  - Området ovanför kan betraktas som ledigt



Lokala variabler och stacken (2)

- Internt används två pekare (*base* och *stack*) för att hålla reda på funktionens del av stacken
  - Området mellan pekarna kallas för *stack frame* (aktiveringspost)
  - Området under kan betraktas som upptaget
  - Området ovanför kan betraktas som ledigt
- När funktionen körs är endast minnet inom *stack frame* åtkomligt
  - Det är en av vinsterna med funktioner, att förändringar kan endast göras lokalt (lokalitet)

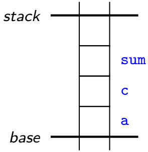


Lokala variabler, minne (1)

code/vars-on-stack.c

```c
int main(void)
{
    int a = 2;
    int c = 3;
    int sum;
```

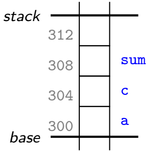
- Varje variabel ligger lagrad på en adress i minnet



Lokala variabler, minne (1)

code/vars-on-stack.c

```c
int main(void)
{
    int a = 2;
    int c = 3;
    int sum;
```

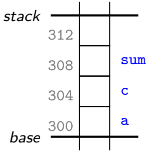


- Varje variabel ligger lagrad på en adress i minnet
- Här ligger
  - a lagrad på adressen 300 (–303)
  - c på adress 304 (–307),
  - sum på adress 308 (–311) och
  - *base*-pekaren har värdet 300
  - (adress 312 är reserverad — mer sen)

Lokala variabler, minne (1)

code/vars-on-stack.c

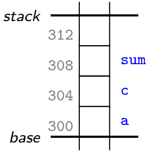
```
int main(void)
{
    int a = 2;
    int c = 3;
    int sum;
```



- Varje variabel ligger lagrad på en adress i minnet
- Här ligger
  - a lagrad på adressen 300 (–303)
  - c på adress 304 (–307),
  - sum på adress 308 (–311) och
  - *base*-pekaren har värdet 300
  - (adress 312 är reserverad — mer sen)
- Kompilatorn översätter variabelreferenser i källkoden till minnesreferenser i maskinkoden

Lokala variabler, minne (1)

code/vars-on-stack.c
```c
int main(void)
{
    int a = 2;
    int c = 3;
    int sum;
```



- Varje variabel ligger lagrad på en adress i minnet
- Här ligger
  - a lagrad på adressen 300 (–303)
  - c på adress 304 (–307),
  - sum på adress 308 (–311) och
  - *base*-pekaren har värdet 300
  - (adress 312 är reserverad — mer sen)
- Kompilatorn översätter variabelreferenser i källkoden till minnesreferenser i maskinkoden
  - Variabeln a översätts till *(base+0)*
  - Variabeln c översätts till *(base+4)*
  - Variabeln sum översätts till *(base+8)*

Lokala variabler, minne (2)

- Mängden minne som kompilatorn reserverar till en variabel bestäms av dess typ
  - En `int` tar vanligen 4 bytes
  - En `char` tar vanligen 1 byte
  - En `double` tar vanligen 8 bytes
- Jag kommer att ignorera det i mina skisser om det inte är viktigt
- Kom ihåg: Ni behöver hålla reda på
  - variabelns namn
  - variabelns typ

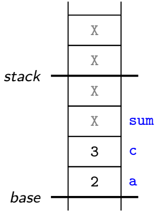
  men inte
  - variabelns adress
  - variabelns storlek

Funktionsanrop (1)

- Stacken används också för parameteröverföring vid funktionsanrop

code/vars-on-stack.c

```c
int main(void)
{
    int a = 2;
    int c = 3;
    int sum;
    sum = add(a, c);
```



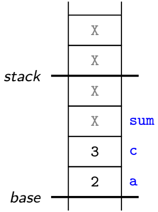
Funktionsanrop (1)

- Stacken används också för parameteröverföring vid funktionsanrop

code/vars-on-stack.c

```
12 int main(void)
13 {
14     int a = 2;
15     int c = 3;
16     int sum;
17     sum = add(a, c);
```

- Koden på rad 17 översätts av kompilatorn till ungefär följande operationer



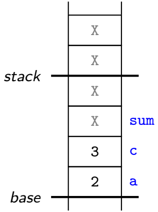
1) Beräkna och lagra en returadress på stacken
2) Evaluera uttrycket för parameter 2 och lägg det på stacken
3) Evaluera uttrycket för parameter 1 och lägg det på stacken
4) Hoppa till funktionen add
5) Ta hand om returvärdet

Funktionsanrop (1)

- Stacken används också för parameteröverföring vid funktionsanrop

code/vars-on-stack.c

```
12 int main(void)
13 {
14     int a = 2;
15     int c = 3;
16     int sum;
17     sum = add(a, c);
```



- Koden på rad 17 översätts av kompilatorn till ungefär följande operationer
- Den översatta maskinkoden kommer att hamna någonstans i minnet

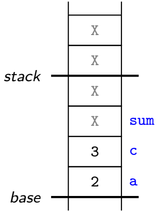
1) Beräkna och lagra en returadress på stacken
2) Evaluera uttrycket för parameter 2 och lägg det på stacken
3) Evaluera uttrycket för parameter 1 och lägg det på stacken
4) Hoppa till funktionen add
5) Ta hand om returvärdet

Funktionsanrop (1)

- Stacken används också för parameteröverföring vid funktionsanrop

code/vars-on-stack.c

```c
12 int main(void)
13 {
14     int a = 2;
15     int c = 3;
16     int sum;
17     sum = add(a, c);
```



- Koden på rad 17 översätts av kompilatorn till ungefär följande operationer
- Den översatta maskinkoden kommer att hamna någonstans i minnet
  - Jag illustrerar med fejkade adresser som börjar på radnumret
    - Varje instruktion antas ta 10 bytes

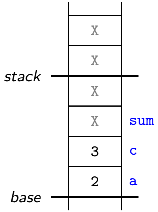
1) Beräkna och lagra en returadress på stacken
2) Evaluera uttrycket för parameter 2 och lägg det på stacken
3) Evaluera uttrycket för parameter 1 och lägg det på stacken
4) Hoppa till funktionen add
5) Ta hand om returvärdet

Funktionsanrop (1)

- Stacken används också för parameteröverföring vid funktionsanrop

code/vars-on-stack.c

```c
12 int main(void)
13 {
14     int a = 2;
15     int c = 3;
16     int sum;
17     sum = add(a, c);
```



- Koden på rad 17 översätts av kompilatorn till ungefär följande operationer
- Den översatta maskinkoden kommer att hamna någonstans i minnet
  - Jag illustrerar med fejkade adresser som börjar på radnumret
    - Varje instruktion antas ta 10 bytes

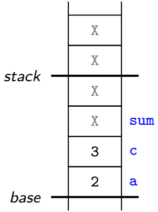
17010	Beräkna och lagra en returadress på stacken
17020	Evaluera uttrycket för parameter 2 och lägg det på stacken
17030	Evaluera uttrycket för parameter 1 och lägg det på stacken
17040	Hoppa till funktionen add
17050	Ta hand om returvärdet

Funktionsanrop (2)

- Stacken används också för parameteröverföring vid funktionsanrop

code/vars-on-stack.c

```
int main(void)
{
    int a = 2;
    int c = 3;
    int sum;
    sum = add(a, c);
```



- I mer detalj så översätts instruktionerna

17010 Beräkna och lagra en returadress på stacken
17020 Evaluera uttrycket för parameter 2 och lägg det på stacken
17030 Evaluera uttrycket för parameter 1 och lägg det på stacken
17040 Hoppa till funktionen add
17050 Ta hand om returvärdet

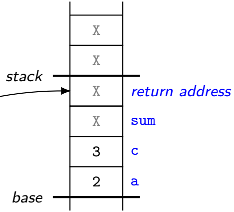
till

17010
17020
17030
17040
17050

Funktionsanrop (2)

- Stacken används också för parameteröverföring vid funktionsanrop

code/vars-on-stack.c
```c
int main(void)
{
    int a = 2;
    int c = 3;
    int sum;
    sum = add(a, c);
```



- I mer detalj så översätts instruktionerna

17010 Beräkna och lagra en returadress på stacken
17020 Evaluera uttrycket för parameter 2 och lägg det på stacken
17030 Evaluera uttrycket för parameter 1 och lägg det på stacken
17040 Hoppa till funktionen add
17050 Ta hand om returvärdet

till

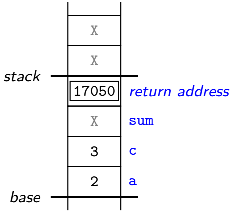
```
17010   store 17050 at (stack-4)     // ret addr
17020
17030
17040
17050
```

Funktionsanrop (2)

- Stacken används också för parameteröverföring vid funktionsanrop

code/vars-on-stack.c

```c
int main(void)
{
    int a = 2;
    int c = 3;
    int sum;
    sum = add(a, c);
```



- I mer detalj så översätts instruktionerna

17010	Beräkna och lagra en returadress på stacken
17020	Evaluera uttrycket för parameter 2 och lägg det på stacken
17030	Evaluera uttrycket för parameter 1 och lägg det på stacken
17040	Hoppa till funktionen add
17050	Ta hand om returvärdet

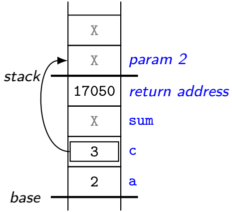
till

```
17010   store 17050 at (stack-4)    // ret addr
17020
17030
17040
17050
```

Funktionsanrop (2)

- Stacken används också för parameteröverföring vid funktionsanrop

code/vars-on-stack.c
```c
int main(void)
{
    int a = 2;
    int c = 3;
    int sum;
    sum = add(a, c);
```



- I mer detalj så översätts instruktionerna

17010	Beräkna och lagra en returadress på stacken
17020	Evaluera uttrycket för parameter 2 och lägg det på stacken
17030	Evaluera uttrycket för parameter 1 och lägg det på stacken
17040	Hoppa till funktionen add
17050	Ta hand om returvärdet

till

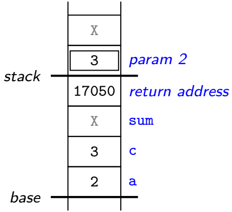
```
17010   store 17050 at (stack-4)    // ret addr
17020   store (base+4) at (stack+0) // c -> p2
17030
17040
17050
```

Funktionsanrop (2)

- Stacken används också för parameteröverföring vid funktionsanrop

code/vars-on-stack.c

```c
int main(void)
{
    int a = 2;
    int c = 3;
    int sum;
    sum = add(a, c);
```



- I mer detalj så översätts instruktionerna

17010	Beräkna och lagra en returadress på stacken
17020	Evaluera uttrycket för parameter 2 och lägg det på stacken
17030	Evaluera uttrycket för parameter 1 och lägg det på stacken
17040	Hoppa till funktionen add
17050	Ta hand om returvärdet

till

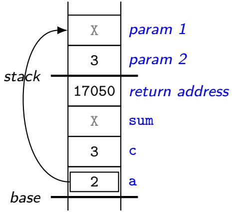
```
17010   store 17050 at (stack-4)    // ret addr
17020   store (base+4) at (stack+0) // c -> p2
17030
17040
17050
```

Funktionsanrop (2)

- Stacken används också för parameteröverföring vid funktionsanrop

code/vars-on-stack.c

```c
int main(void)
{
    int a = 2;
    int c = 3;
    int sum;
    sum = add(a, c);
```



- I mer detalj så översätts instruktionerna

17010 Beräkna och lagra en returadress på stacken
17020 Evaluera uttrycket för parameter 2 och lägg det på stacken
17030 Evaluera uttrycket för parameter 1 och lägg det på stacken
17040 Hoppa till funktionen add
17050 Ta hand om returvärdet

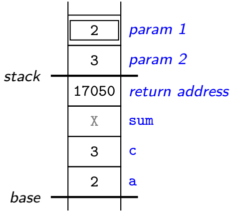
till

```
17010   store 17050 at (stack-4)    // ret addr
17020   store (base+4) at (stack+0) // c -> p2
17030   store (base+0) at (stack+4) // a -> p1
17040
17050
```

Funktionsanrop (2)

- Stacken används också för parameteröverföring vid funktionsanrop

code/vars-on-stack.c
```c
int main(void)
{
    int a = 2;
    int c = 3;
    int sum;
    sum = add(a, c);
```



- I mer detalj så översätts instruktionerna

17010	Beräkna och lagra en returadress på stacken
17020	Evaluera uttrycket för parameter 2 och lägg det på stacken
17030	Evaluera uttrycket för parameter 1 och lägg det på stacken
17040	Hoppa till funktionen add
17050	Ta hand om returvärdet

till

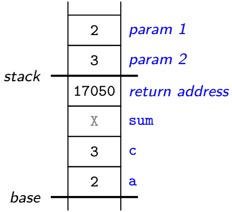
```
17010   store 17050 at (stack-4)    // ret addr
17020   store (base+4) at (stack+0) // c -> p2
17030   store (base+0) at (stack+4) // a -> p1
17040
17050
```

Funktionsanrop (2)

- Stacken används också för parameteröverföring vid funktionsanrop

code/vars-on-stack.c

```c
int main(void)
{
    int a = 2;
    int c = 3;
    int sum;
    sum = add(a, c);
```



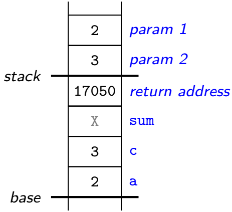
- I mer detalj så översätts instruktionerna

17010	Beräkna och lagra en returadress på stacken
17020	Evaluera uttrycket för parameter 2 och lägg det på stacken
17030	Evaluera uttrycket för parameter 1 och lägg det på stacken
17040	Hoppa till funktionen add
17050	Ta hand om returvärdet

till

```
17010   store 17050 at (stack-4)    // ret addr
17020   store (base+4) at (stack+0) // c -> p2
17030   store (base+0) at (stack+4) // a -> p1
17040   call add
17050
```

Funktionsanrop (2)

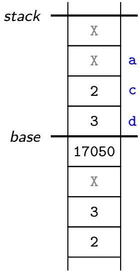


Funktionsanrop (3)

- Den anropade funktionen (`add()`) justerar stackpekarna...

code/vars-on-stack.c

```c
int add(int c, int d)
{
    int a;
    a = c + d;
    return a;
}
```

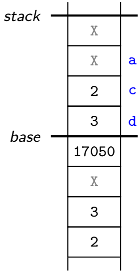


Funktionsanrop (3)

- Den anropade funktionen (`add()`) justerar stackpekarna. . .

code/vars-on-stack.c

```c
int add(int c, int d)
{
    int a;
    a = c + d;
    return a;
}
```



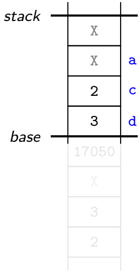
- . . . och reserverar en egen *stack frame* i den fria delen av stacken

Funktionsanrop (3)

- Den anropade funktionen (`add()`) justerar stackpekarna...

code/vars-on-stack.c

```c
int add(int c, int d)
{
    int a;
    a = c + d;
    return a;
}
```



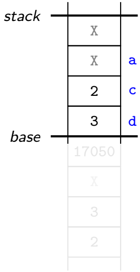
- ...och reserverar en egen *stack frame* i den fria delen av stacken
- Den anropande funktionens variabler blir osynliga och skyddas

Funktionsanrop (3)

- Den anropade funktionen (`add()`) justerar stackpekarna...

code/vars-on-stack.c

```c
int add(int c, int d)
{
    int a;
    a = c + d;
    return a;
}
```



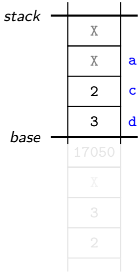
- ...och reserverar en egen *stack frame* i den fria delen av stacken
- Den anropande funktionens variabler blir osynliga och skyddas
- Notera att parametrarna `c` och `d` fungerar som lokala variabler

Funktionsanrop (3)

- Den anropade funktionen (`add()`) justerar stackpekarna...

code/vars-on-stack.c

```c
int add(int c, int d)
{
    int a;
    a = c + d;
    return a;
}
```



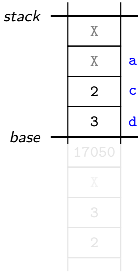
- ...och reserverar en egen *stack frame* i den fria delen av stacken
- Den anropande funktionens variabler blir osynliga och skyddas
- Notera att parametrarna `c` och `d` fungerar som lokala variabler
  - Parametrarna är initierade (har giltiga värden) när funktionen startar

Funktionsanrop (3)

- Den anropade funktionen (`add()`) justerar stackpekarna...

code/vars-on-stack.c

```c
int add(int c, int d)
{
    int a;
    a = c + d;
    return a;
}
```

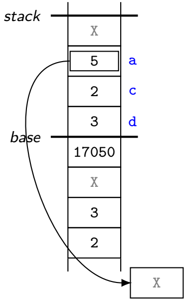


- ...och reserverar en egen *stack frame* i den fria delen av stacken
- Den anropande funktionens variabler blir osynliga och skyddas
- Notera att parametrarna `c` och `d` fungerar som lokala variabler
  - Parametrarna är initierade (har giltiga värden) när funktionen startar
  - Övriga lokala variabler har ett odefinierat värde (X i stacken)

Återhopp

- Vid återhopp från en funktion sker följande:
  - returvärdet lagras i ett s.k. register i CPU:n,

code/vars-on-stack.c
```c
int add(int c, int d)
{
    int a;
    a = c + d;
    return a;
}
int main(void)
{
    int a = 2;
    int c = 3;
    int sum;
    sum = add(a, c);
```

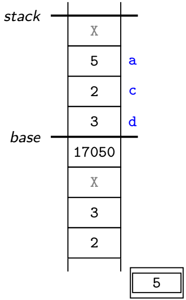


Återhopp

- Vid återhopp från en funktion sker följande:
  - returvärdet lagras i ett s.k. register i CPU:n,

code/vars-on-stack.c

```c
int add(int c, int d)
{
    int a;
    a = c + d;
    return a;
}
int main(void)
{
    int a = 2;
    int c = 3;
    int sum;
    sum = add(a, c);
```

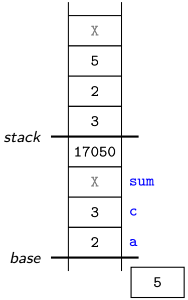


Återhopp

- Vid återhopp från en funktion sker följande:
  - returvärdet lagras i ett s.k. register i CPU:n,
  - *stack frame* återställs. . .

code/vars-on-stack.c

```c
int add(int c, int d)
{
    int a;
    a = c + d;
    return a;
}
int main(void)
{
    int a = 2;
    int c = 3;
    int sum;
    sum = add(a, c);
```

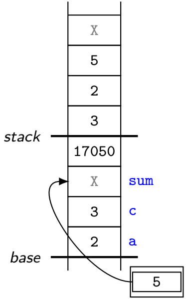


Återhopp

- Vid återhopp från en funktion sker följande:
  - returvärdet lagras i ett s.k. register i CPU:n,
  - *stack frame* återställs. . .

code/vars-on-stack.c

```c
int add(int c, int d)
{
    int a;
    a = c + d;
    return a;
}
int main(void)
{
    int a = 2;
    int c = 3;
    int sum;
    sum = add(a, c);
```



- . . . exekveringen fortsätter vid återhoppsadressen. . .

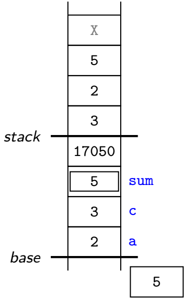
```
17050    store register at (base+8) // sum
```

Återhopp

- Vid återhopp från en funktion sker följande:
  - returvärdet lagras i ett s.k. register i CPU:n,
  - *stack frame* återställs. . .

code/vars-on-stack.c

```c
int add(int c, int d)
{
    int a;
    a = c + d;
    return a;
}
int main(void)
{
    int a = 2;
    int c = 3;
    int sum;
    sum = add(a, c);
```



- . . . exekveringen fortsätter vid återhoppsadressen. . .

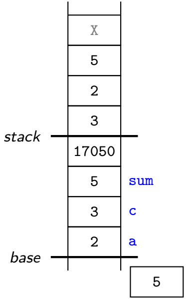
```
17050    store register at (base+8) // sum
```

Återhopp

- Vid återhopp från en funktion sker följande:
  - returvärdet lagras i ett s.k. register i CPU:n,
  - *stack frame* återställs. . .

code/vars-on-stack.c

```
int add(int c, int d)
{
    int a;
    a = c + d;
    return a;
}
int main(void)
{
    int a = 2;
    int c = 3;
    int sum;
    sum = add(a, c);
```



- . . . exekveringen fortsätter vid återhoppsadressen. . .

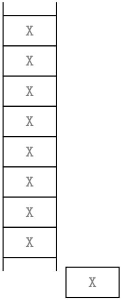
```
17050   store register at (base+8) // sum
```

- . . . och sedan vidare på nästa rad. . .

Exemplet steg för steg

code/vars-on-stack.c

```c
#include <stdio.h>
void print(int v)
{
    printf("v = %d\n",v);
}
int add(int c, int d)
{
    int a;
    a = c + d;
    return a;
}
int main(void)
{
    int a = 2;
    int c = 3;
    int sum;
    sum = add(a, c);
    print(sum);
    sum = add(sum, c + 4);
    print(sum);
    return 0;
}
```

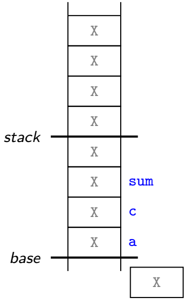


[redacted]

Exemplet steg för steg

code/vars-on-stack.c

```
#include <stdio.h>
void print(int v)
{
    printf("v = %d\n",v);
}
int add(int c, int d)
{
    int a;
    a = c + d;
    return a;
}
int main(void)
{
    int a = 2;
    int c = 3;
    int sum;
    sum = add(a, c);
    print(sum);
    sum = add(sum, c + 4);
    print(sum);
    return 0;
}
```

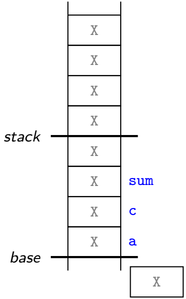


[redacted]

Exemplet steg för steg

code/vars-on-stack.c

```c
#include <stdio.h>
void print(int v)
{
    printf("v = %d\n",v);
}
int add(int c, int d)
{
    int a;
    a = c + d;
    return a;
}
int main(void)
{
    int a = 2;
    int c = 3;
    int sum;
    sum = add(a, c);
    print(sum);
    sum = add(sum, c + 4);
    print(sum);
    return 0;
}
```

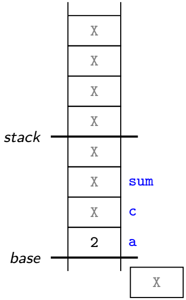


[redacted]

Exemplet steg för steg

code/vars-on-stack.c

```c
#include <stdio.h>
void print(int v)
{
    printf("v = %d\n",v);
}
int add(int c, int d)
{
    int a;
    a = c + d;
    return a;
}
int main(void)
{
    int a = 2;
    int c = 3;
    int sum;
    sum = add(a, c);
    print(sum);
    sum = add(sum, c + 4);
    print(sum);
    return 0;
}
```

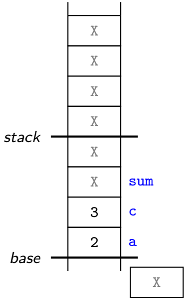


[redacted]

Exemplet steg för steg

code/vars-on-stack.c

```c
#include <stdio.h>
void print(int v)
{
    printf("v = %d\n",v);
}
int add(int c, int d)
{
    int a;
    a = c + d;
    return a;
}
int main(void)
{
    int a = 2;
    int c = 3;
    int sum;
    sum = add(a, c);
    print(sum);
    sum = add(sum, c + 4);
    print(sum);
    return 0;
}
```

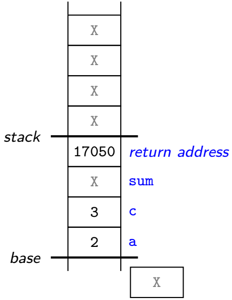


[redacted]

Exemplet steg för steg

code/vars-on-stack.c

```c
#include <stdio.h>
void print(int v)
{
    printf("v = %d\n",v);
}
int add(int c, int d)
{
    int a;
    a = c + d;
    return a;
}
int main(void)
{
    int a = 2;
    int c = 3;
    int sum;
    sum = add(a, c);
    print(sum);
    sum = add(sum, c + 4);
    print(sum);
    return 0;
}
```

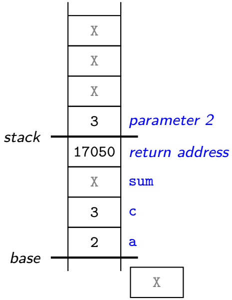


[redacted]

Exemplet steg för steg

code/vars-on-stack.c

```c
#include <stdio.h>
void print(int v)
{
    printf("v = %d\n",v);
}
int add(int c, int d)
{
    int a;
    a = c + d;
    return a;
}
int main(void)
{
    int a = 2;
    int c = 3;
    int sum;
    sum = add(a, c);
    print(sum);
    sum = add(sum, c + 4);
    print(sum);
    return 0;
}
```

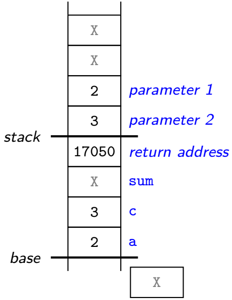


[redacted]

Exemplet steg för steg

code/vars-on-stack.c

```c
#include <stdio.h>
void print(int v)
{
    printf("v = %d\n",v);
}
int add(int c, int d)
{
    int a;
    a = c + d;
    return a;
}
int main(void)
{
    int a = 2;
    int c = 3;
    int sum;
    sum = add(a, c);
    print(sum);
    sum = add(sum, c + 4);
    print(sum);
    return 0;
}
```

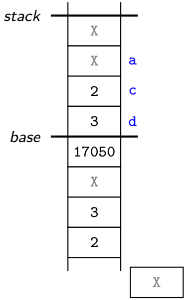


[redacted]

Exemplet steg för steg

code/vars-on-stack.c

```c
#include <stdio.h>
void print(int v)
{
    printf("v = %d\n",v);
}
int add(int c, int d)
{
    int a;
    a = c + d;
    return a;
}
int main(void)
{
    int a = 2;
    int c = 3;
    int sum;
    sum = add(a, c);
    print(sum);
    sum = add(sum, c + 4);
    print(sum);
    return 0;
}
```

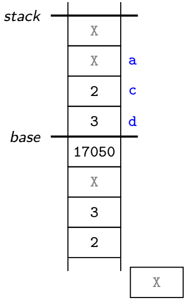


[redacted]

Exemplet steg för steg

code/vars-on-stack.c

```c
#include <stdio.h>
void print(int v)
{
    printf("v = %d\n",v);
}
int add(int c, int d)
{
    int a;
    a = c + d;
    return a;
}
int main(void)
{
    int a = 2;
    int c = 3;
    int sum;
    sum = add(a, c);
    print(sum);
    sum = add(sum, c + 4);
    print(sum);
    return 0;
}
```

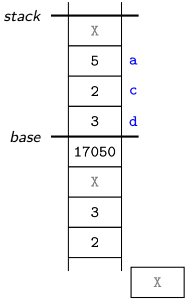


[redacted]

Exemplet steg för steg

code/vars-on-stack.c

```c
#include <stdio.h>
void print(int v)
{
    printf("v = %d\n",v);
}
int add(int c, int d)
{
    int a;
    a = c + d;
    return a;
}
int main(void)
{
    int a = 2;
    int c = 3;
    int sum;
    sum = add(a, c);
    print(sum);
    sum = add(sum, c + 4);
    print(sum);
    return 0;
}
```

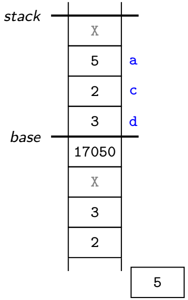


[redacted]

Exemplet steg för steg

code/vars-on-stack.c

```c
#include <stdio.h>
void print(int v)
{
    printf("v = %d\n",v);
}
int add(int c, int d)
{
    int a;
    a = c + d;
    return a;
}
int main(void)
{
    int a = 2;
    int c = 3;
    int sum;
    sum = add(a, c);
    print(sum);
    sum = add(sum, c + 4);
    print(sum);
    return 0;
}
```

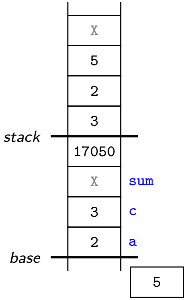


[redacted]

Exemplet steg för steg

code/vars-on-stack.c

```c
#include <stdio.h>
void print(int v)
{
    printf("v = %d\n",v);
}
int add(int c, int d)
{
    int a;
    a = c + d;
    return a;
}
int main(void)
{
    int a = 2;
    int c = 3;
    int sum;
    sum = add(a, c);
    print(sum);
    sum = add(sum, c + 4);
    print(sum);
    return 0;
}
```

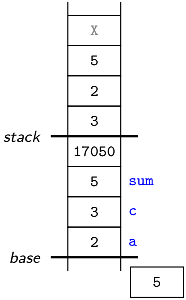


[redacted]

Exemplet steg för steg

code/vars-on-stack.c

```c
#include <stdio.h>
void print(int v)
{
    printf("v = %d\n",v);
}
int add(int c, int d)
{
    int a;
    a = c + d;
    return a;
}
int main(void)
{
    int a = 2;
    int c = 3;
    int sum;
    sum = add(a, c);
    print(sum);
    sum = add(sum, c + 4);
    print(sum);
    return 0;
}
```

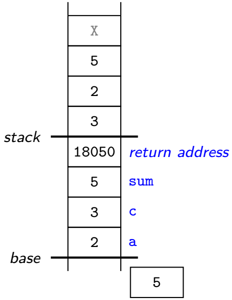


[redacted]

Exemplet steg för steg

code/vars-on-stack.c

```c
#include <stdio.h>
void print(int v)
{
    printf("v = %d\n",v);
}
int add(int c, int d)
{
    int a;
    a = c + d;
    return a;
}
int main(void)
{
    int a = 2;
    int c = 3;
    int sum;
    sum = add(a, c);
    print(sum);
    sum = add(sum, c + 4);
    print(sum);
    return 0;
}
```

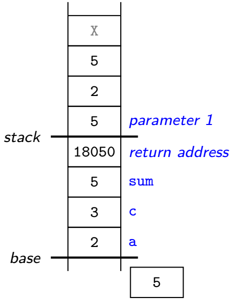


[redacted]

Exemplet steg för steg

code/vars-on-stack.c

```c
#include <stdio.h>
void print(int v)
{
    printf("v = %d\n",v);
}
int add(int c, int d)
{
    int a;
    a = c + d;
    return a;
}
int main(void)
{
    int a = 2;
    int c = 3;
    int sum;
    sum = add(a, c);
    print(sum);
    sum = add(sum, c + 4);
    print(sum);
    return 0;
}
```

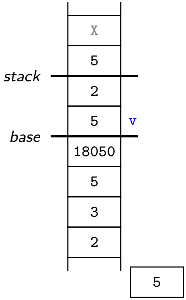


[redacted]

Exemplet steg för steg

code/vars-on-stack.c

```c
#include <stdio.h>
void print(int v)
{
    printf("v = %d\n",v);
}
int add(int c, int d)
{
    int a;
    a = c + d;
    return a;
}
int main(void)
{
    int a = 2;
    int c = 3;
    int sum;
    sum = add(a, c);
    print(sum);
    sum = add(sum, c + 4);
    print(sum);
    return 0;
}
```

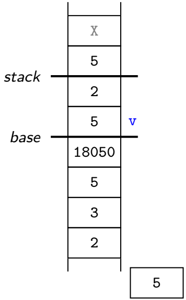


[redacted]

Exemplet steg för steg

code/vars-on-stack.c

```c
#include <stdio.h>
void print(int v)
{
    printf("v = %d\n",v);
}
int add(int c, int d)
{
    int a;
    a = c + d;
    return a;
}
int main(void)
{
    int a = 2;
    int c = 3;
    int sum;
    sum = add(a, c);
    print(sum);
    sum = add(sum, c + 4);
    print(sum);
    return 0;
}
```

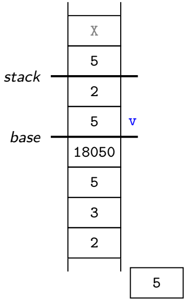


[redacted]

Exemplet steg för steg

code/vars-on-stack.c

```c
#include <stdio.h>
void print(int v)
{
    printf("v = %d\n",v);
}
int add(int c, int d)
{
    int a;
    a = c + d;
    return a;
}
int main(void)
{
    int a = 2;
    int c = 3;
    int sum;
    sum = add(a, c);
    print(sum);
    sum = add(sum, c + 4);
    print(sum);
    return 0;
}
```

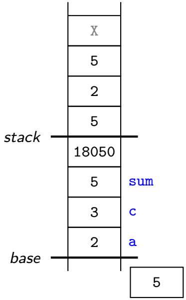



```
v = 5
```

Exemplet steg för steg

code/vars-on-stack.c

```
#include <stdio.h>
void print(int v)
{
    printf("v = %d\n",v);
}
int add(int c, int d)
{
    int a;
    a = c + d;
    return a;
}
int main(void)
{
    int a = 2;
    int c = 3;
    int sum;
    sum = add(a, c);
    print(sum);
    sum = add(sum, c + 4);
    print(sum);
    return 0;
}
```

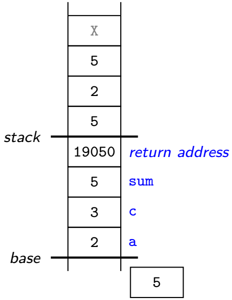



```
v = 5
```

Exemplet steg för steg

code/vars-on-stack.c

```c
#include <stdio.h>
void print(int v)
{
    printf("v = %d\n",v);
}
int add(int c, int d)
{
    int a;
    a = c + d;
    return a;
}
int main(void)
{
    int a = 2;
    int c = 3;
    int sum;
    sum = add(a, c);
    print(sum);
    sum = add(sum, c + 4);
    print(sum);
    return 0;
}
```

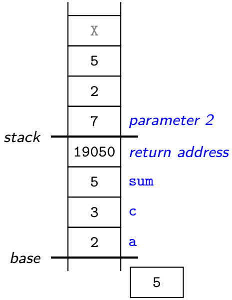



```
v = 5
```

Exemplet steg för steg

code/vars-on-stack.c

```c
#include <stdio.h>
void print(int v)
{
    printf("v = %d\n",v);
}
int add(int c, int d)
{
    int a;
    a = c + d;
    return a;
}
int main(void)
{
    int a = 2;
    int c = 3;
    int sum;
    sum = add(a, c);
    print(sum);
    sum = add(sum, c + 4);
    print(sum);
    return 0;
}
```

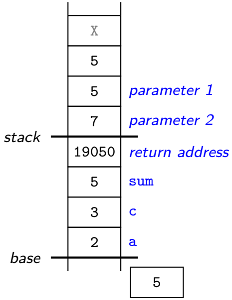



```
v = 5
```

Exemplet steg för steg

code/vars-on-stack.c

```c
#include <stdio.h>
void print(int v)
{
    printf("v = %d\n",v);
}
int add(int c, int d)
{
    int a;
    a = c + d;
    return a;
}
int main(void)
{
    int a = 2;
    int c = 3;
    int sum;
    sum = add(a, c);
    print(sum);
    sum = add(sum, c + 4);
    print(sum);
    return 0;
}
```

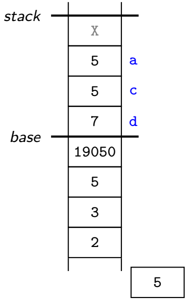



```
v = 5
```

Exemplet steg för steg

code/vars-on-stack.c

```c
#include <stdio.h>
void print(int v)
{
    printf("v = %d\n",v);
}
int add(int c, int d)
{
    int a;
    a = c + d;
    return a;
}
int main(void)
{
    int a = 2;
    int c = 3;
    int sum;
    sum = add(a, c);
    print(sum);
    sum = add(sum, c + 4);
    print(sum);
    return 0;
}
```

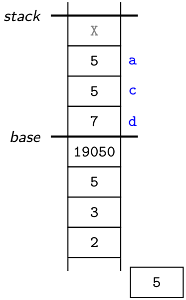



```
v = 5
```

Exemplet steg för steg

code/vars-on-stack.c

```c
#include <stdio.h>
void print(int v)
{
    printf("v = %d\n",v);
}
int add(int c, int d)
{
    int a;
    a = c + d;
    return a;
}
int main(void)
{
    int a = 2;
    int c = 3;
    int sum;
    sum = add(a, c);
    print(sum);
    sum = add(sum, c + 4);
    print(sum);
    return 0;
}
```

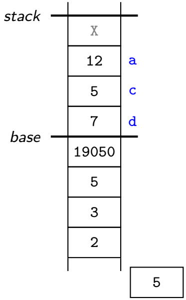



```
v = 5
```

Exemplet steg för steg

code/vars-on-stack.c

```c
#include <stdio.h>
void print(int v)
{
    printf("v = %d\n",v);
}
int add(int c, int d)
{
    int a;
    a = c + d;
    return a;
}
int main(void)
{
    int a = 2;
    int c = 3;
    int sum;
    sum = add(a, c);
    print(sum);
    sum = add(sum, c + 4);
    print(sum);
    return 0;
}
```

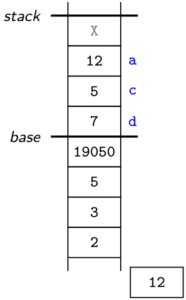



```
v = 5
```

Exemplet steg för steg

code/vars-on-stack.c

```c
#include <stdio.h>
void print(int v)
{
    printf("v = %d\n",v);
}
int add(int c, int d)
{
    int a;
    a = c + d;
    return a;
}
int main(void)
{
    int a = 2;
    int c = 3;
    int sum;
    sum = add(a, c);
    print(sum);
    sum = add(sum, c + 4);
    print(sum);
    return 0;
}
```

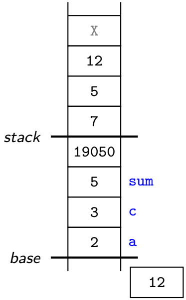



```
v = 5
```

Exemplet steg för steg

code/vars-on-stack.c

```c
#include <stdio.h>
void print(int v)
{
    printf("v = %d\n",v);
}
int add(int c, int d)
{
    int a;
    a = c + d;
    return a;
}
int main(void)
{
    int a = 2;
    int c = 3;
    int sum;
    sum = add(a, c);
    print(sum);
    sum = add(sum, c + 4);
    print(sum);
    return 0;
}
```

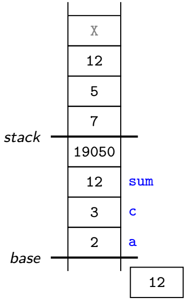



```
v = 5
```

Exemplet steg för steg

code/vars-on-stack.c

```c
#include <stdio.h>
void print(int v)
{
    printf("v = %d\n",v);
}
int add(int c, int d)
{
    int a;
    a = c + d;
    return a;
}
int main(void)
{
    int a = 2;
    int c = 3;
    int sum;
    sum = add(a, c);
    print(sum);
    sum = add(sum, c + 4);
    print(sum);
    return 0;
}
```

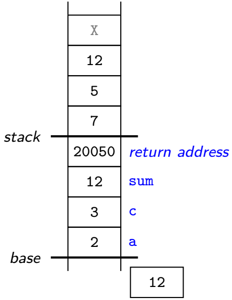



```
v = 5
```

Exemplet steg för steg

code/vars-on-stack.c

```c
#include <stdio.h>
void print(int v)
{
    printf("v = %d\n",v);
}
int add(int c, int d)
{
    int a;
    a = c + d;
    return a;
}
int main(void)
{
    int a = 2;
    int c = 3;
    int sum;
    sum = add(a, c);
    print(sum);
    sum = add(sum, c + 4);
    print(sum);
    return 0;
}
```



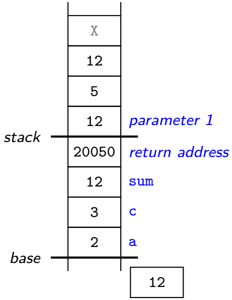

```
v = 5
```

Exemplet steg för steg

code/vars-on-stack.c

```c
#include <stdio.h>
void print(int v)
{
    printf("v = %d\n",v);
}
int add(int c, int d)
{
    int a;
    a = c + d;
    return a;
}
int main(void)
{
    int a = 2;
    int c = 3;
    int sum;
    sum = add(a, c);
    print(sum);
    sum = add(sum, c + 4);
    print(sum);
    return 0;
}
```

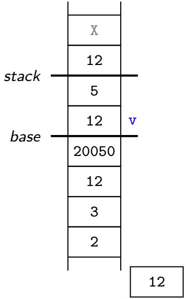
```
v = 5
```



Exemplet steg för steg

code/vars-on-stack.c

```c
#include <stdio.h>
void print(int v)
{
    printf("v = %d\n",v);
}
int add(int c, int d)
{
    int a;
    a = c + d;
    return a;
}
int main(void)
{
    int a = 2;
    int c = 3;
    int sum;
    sum = add(a, c);
    print(sum);
    sum = add(sum, c + 4);
    print(sum);
    return 0;
}
```

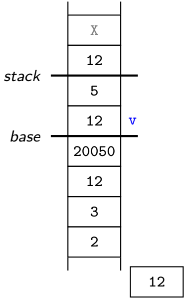



```
v = 5
```

Exemplet steg för steg

code/vars-on-stack.c

```c
#include <stdio.h>
void print(int v)
{
    printf("v = %d\n",v);
}
int add(int c, int d)
{
    int a;
    a = c + d;
    return a;
}
int main(void)
{
    int a = 2;
    int c = 3;
    int sum;
    sum = add(a, c);
    print(sum);
    sum = add(sum, c + 4);
    print(sum);
    return 0;
}
```

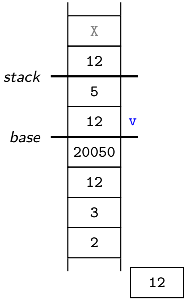



```
v = 5
```

Exemplet steg för steg

code/vars-on-stack.c

```c
#include <stdio.h>
void print(int v)
{
    printf("v = %d\n",v);
}
int add(int c, int d)
{
    int a;
    a = c + d;
    return a;
}
int main(void)
{
    int a = 2;
    int c = 3;
    int sum;
    sum = add(a, c);
    print(sum);
    sum = add(sum, c + 4);
    print(sum);
    return 0;
}
```

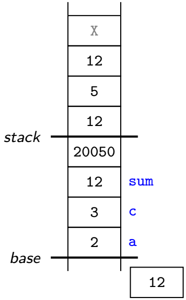



```
v = 5
v = 12
```

Exemplet steg för steg

code/vars-on-stack.c

```c
#include <stdio.h>
void print(int v)
{
    printf("v = %d\n",v);
}
int add(int c, int d)
{
    int a;
    a = c + d;
    return a;
}
int main(void)
{
    int a = 2;
    int c = 3;
    int sum;
    sum = add(a, c);
    print(sum);
    sum = add(sum, c + 4);
    print(sum);
    return 0;
}
```

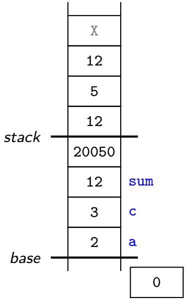



```
v = 5
v = 12
```

Exemplet steg för steg

code/vars-on-stack.c

```c
#include <stdio.h>
void print(int v)
{
    printf("v = %d\n",v);
}
int add(int c, int d)
{
    int a;
    a = c + d;
    return a;
}
int main(void)
{
    int a = 2;
    int c = 3;
    int sum;
    sum = add(a, c);
    print(sum);
    sum = add(sum, c + 4);
    print(sum);
    return 0;
}
```

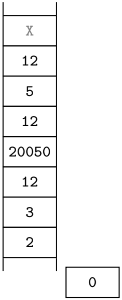



```
v = 5
v = 12
```

Exemplet steg för steg

code/vars-on-stack.c

```c
#include <stdio.h>
void print(int v)
{
    printf("v = %d\n",v);
}
int add(int c, int d)
{
    int a;
    a = c + d;
    return a;
}
int main(void)
{
    int a = 2;
    int c = 3;
    int sum;
    sum = add(a, c);
    print(sum);
    sum = add(sum, c + 4);
    print(sum);
    return 0;
}
```




```
v = 5
v = 12
```

Rekursion (1)

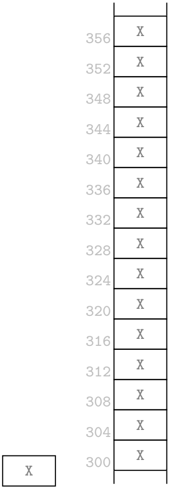
- Här är ett rekursivt exempel som beräknar Fibonacci-talen som definieras

$$F_0 = 0,\ F_1 = 1,\ F_n = F_{n-1} + F_{n-2}$$

```c
#include <stdio.h>

int fib(int n)
{
    int fm1, fm2;
    if (n < 2) {
        return n;
    } else {
        fm1 = fib(n - 1);
        fm2 = fib(n - 2);
        return fm1 + fm2;
    }
}

int main(void)
{
    int f = fib(3);
    printf("fib(3) = %d\n", f);
    return 0;
}
```

Rekursion (2)

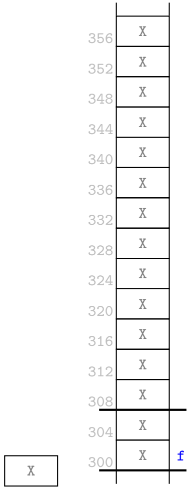
```c
#include <stdio.h>

int fib(int n)
{
    int fm1, fm2;
    if (n < 2) {
        return n;
    } else {
        fm1 = fib(n - 1);
        fm2 = fib(n - 2);
        return fm1 + fm2;
    }
}

int main(void)
{
    int f = fib(3);
    printf("fib(3) = %d\n", f);
    return 0;
}
```



[redacted]

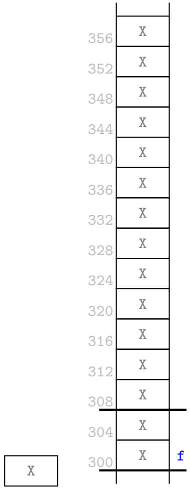
Rekursion (2)

```c
#include <stdio.h>

int fib(int n)
{
    int fm1, fm2;
    if (n < 2) {
        return n;
    } else {
        fm1 = fib(n - 1);
        fm2 = fib(n - 2);
        return fm1 + fm2;
    }
}

int main(void)
{
    int f = fib(3);
    printf("fib(3) = %d\n", f);
    return 0;
}
```



Rekursion (2)

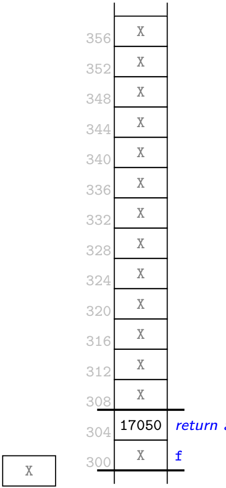
```c
#include <stdio.h>

int fib(int n)
{
    int fm1, fm2;
    if (n < 2) {
        return n;
    } else {
        fm1 = fib(n - 1);
        fm2 = fib(n - 2);
        return fm1 + fm2;
    }
}

int main(void)
{
    int f = fib(3);
    printf("fib(3) = %d\n", f);
    return 0;
}
```



[redacted]

Rekursion (2)

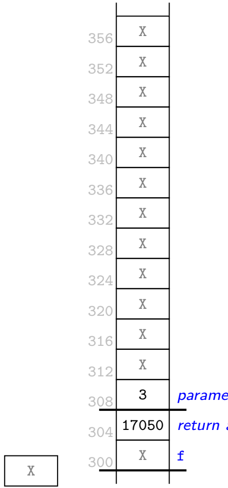
```c
#include <stdio.h>

int fib(int n)
{
    int fm1, fm2;
    if (n < 2) {
        return n;
    } else {
        fm1 = fib(n - 1);
        fm2 = fib(n - 2);
        return fm1 + fm2;
    }
}

int main(void)
{
    int f = fib(3);
    printf("fib(3) = %d\n", f);
    return 0;
}
```



[redacted]

Rekursion (2)

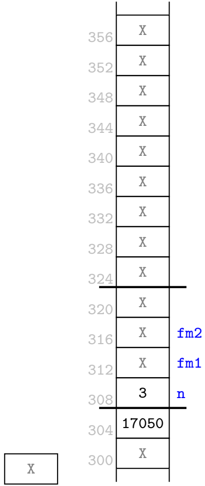
```c
#include <stdio.h>

int fib(int n)
{
    int fm1, fm2;
    if (n < 2) {
        return n;
    } else {
        fm1 = fib(n - 1);
        fm2 = fib(n - 2);
        return fm1 + fm2;
    }
}

int main(void)
{
    int f = fib(3);
    printf("fib(3) = %d\n", f);
    return 0;
}
```



[redacted]

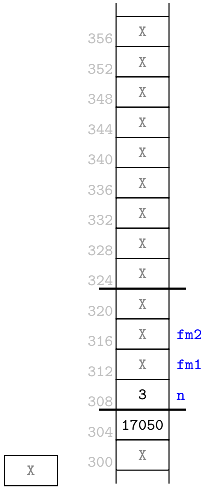
Rekursion (2)

```c
#include <stdio.h>

int fib(int n)
{
    int fm1, fm2;
    if (n < 2) {
        return n;
    } else {
        fm1 = fib(n - 1);
        fm2 = fib(n - 2);
        return fm1 + fm2;
    }
}

int main(void)
{
    int f = fib(3);
    printf("fib(3) = %d\n", f);
    return 0;
}
```



Rekursion (2)

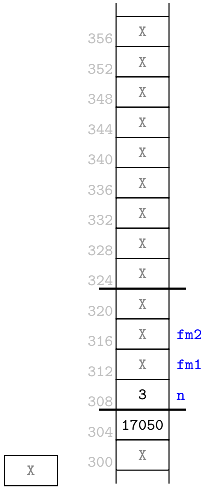
```c
#include <stdio.h>

int fib(int n)
{
    int fm1, fm2;
    if (n < 2) {
        return n;
    } else {
        fm1 = fib(n - 1);
        fm2 = fib(n - 2);
        return fm1 + fm2;
    }
}

int main(void)
{
    int f = fib(3);
    printf("fib(3) = %d\n", f);
    return 0;
}
```



[redacted]

Rekursion (2)

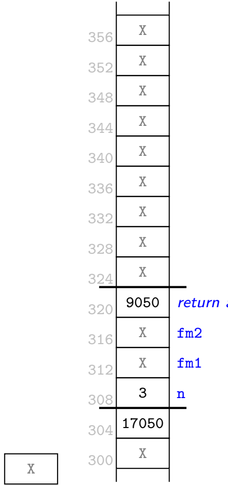
```c
#include <stdio.h>

int fib(int n)
{
    int fm1, fm2;
    if (n < 2) {
        return n;
    } else {
        fm1 = fib(n - 1);
        fm2 = fib(n - 2);
        return fm1 + fm2;
    }
}

int main(void)
{
    int f = fib(3);
    printf("fib(3) = %d\n", f);
    return 0;
}
```



[redacted]

Rekursion (2)

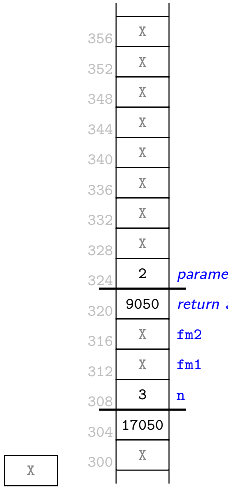
```c
#include <stdio.h>

int fib(int n)
{
    int fm1, fm2;
    if (n < 2) {
        return n;
    } else {
        fm1 = fib(n - 1);
        fm2 = fib(n - 2);
        return fm1 + fm2;
    }
}

int main(void)
{
    int f = fib(3);
    printf("fib(3) = %d\n", f);
    return 0;
}
```



[redacted]

Rekursion (2)

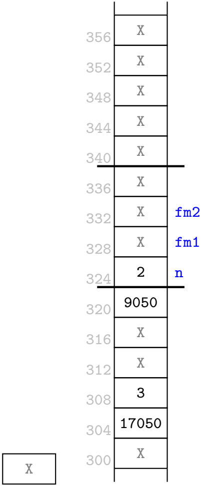
```c
#include <stdio.h>

int fib(int n)
{
    int fm1, fm2;
    if (n < 2) {
        return n;
    } else {
        fm1 = fib(n - 1);
        fm2 = fib(n - 2);
        return fm1 + fm2;
    }
}

int main(void)
{
    int f = fib(3);
    printf("fib(3) = %d\n", f);
    return 0;
}
```



[redacted]

Rekursion (2)

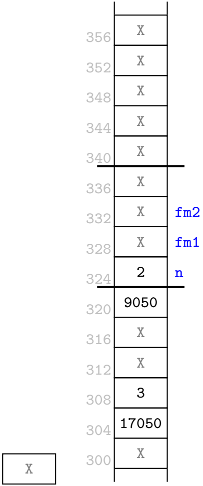
```c
#include <stdio.h>

int fib(int n)
{
    int fm1, fm2;
    if (n < 2) {
        return n;
    } else {
        fm1 = fib(n - 1);
        fm2 = fib(n - 2);
        return fm1 + fm2;
    }
}

int main(void)
{
    int f = fib(3);
    printf("fib(3) = %d\n", f);
    return 0;
}
```



[redacted]

Rekursion (2)

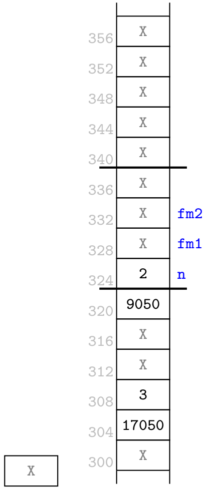
```c
#include <stdio.h>

int fib(int n)
{
    int fm1, fm2;
    if (n < 2) {
        return n;
    } else {
        fm1 = fib(n - 1);
        fm2 = fib(n - 2);
        return fm1 + fm2;
    }
}

int main(void)
{
    int f = fib(3);
    printf("fib(3) = %d\n", f);
    return 0;
}
```



[redacted]

Rekursion (2)

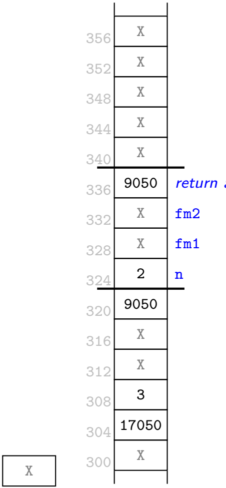
```c
#include <stdio.h>

int fib(int n)
{
    int fm1, fm2;
    if (n < 2) {
        return n;
    } else {
        fm1 = fib(n - 1);
        fm2 = fib(n - 2);
        return fm1 + fm2;
    }
}

int main(void)
{
    int f = fib(3);
    printf("fib(3) = %d\n", f);
    return 0;
}
```



[redacted]

Rekursion (2)

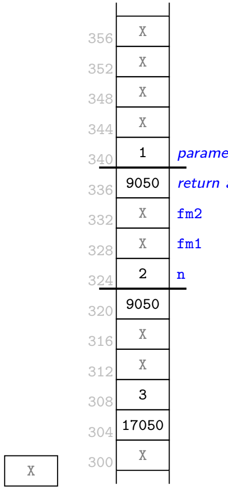
```c
#include <stdio.h>

int fib(int n)
{
    int fm1, fm2;
    if (n < 2) {
        return n;
    } else {
        fm1 = fib(n - 1);
        fm2 = fib(n - 2);
        return fm1 + fm2;
    }
}

int main(void)
{
    int f = fib(3);
    printf("fib(3) = %d\n", f);
    return 0;
}
```



[redacted]

Rekursion (2)

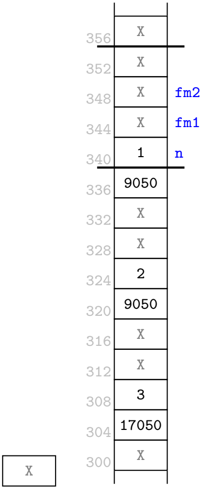
```c
#include <stdio.h>

int fib(int n)
{
    int fm1, fm2;
    if (n < 2) {
        return n;
    } else {
        fm1 = fib(n - 1);
        fm2 = fib(n - 2);
        return fm1 + fm2;
    }
}

int main(void)
{
    int f = fib(3);
    printf("fib(3) = %d\n", f);
    return 0;
}
```



[redacted]

Rekursion (2)

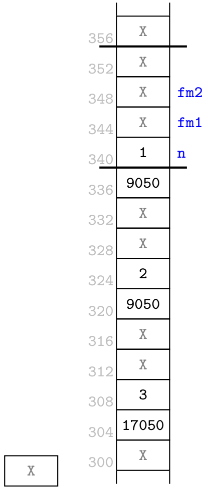
```c
#include <stdio.h>

int fib(int n)
{
    int fm1, fm2;
    if (n < 2) {
        return n;
    } else {
        fm1 = fib(n - 1);
        fm2 = fib(n - 2);
        return fm1 + fm2;
    }
}

int main(void)
{
    int f = fib(3);
    printf("fib(3) = %d\n", f);
    return 0;
}
```



[redacted]

Rekursion (2)

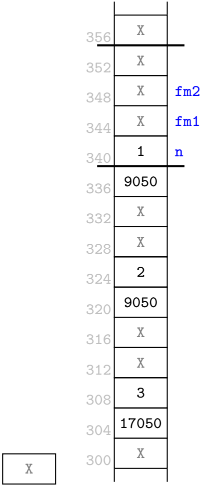
```c
#include <stdio.h>

int fib(int n)
{
    int fm1, fm2;
    if (n < 2) {
        return n;
    } else {
        fm1 = fib(n - 1);
        fm2 = fib(n - 2);
        return fm1 + fm2;
    }
}

int main(void)
{
    int f = fib(3);
    printf("fib(3) = %d\n", f);
    return 0;
}
```



[redacted]

Rekursion (2)

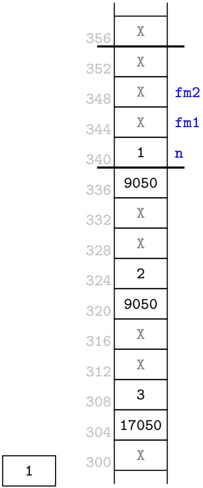
```c
#include <stdio.h>

int fib(int n)
{
    int fm1, fm2;
    if (n < 2) {
        return n;
    } else {
        fm1 = fib(n - 1);
        fm2 = fib(n - 2);
        return fm1 + fm2;
    }
}

int main(void)
{
    int f = fib(3);
    printf("fib(3) = %d\n", f);
    return 0;
}
```



[redacted]

Rekursion (2)

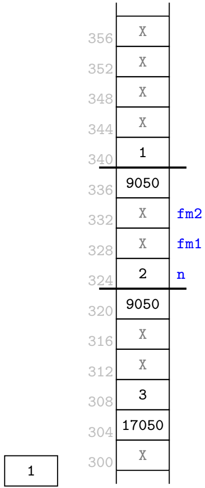
```c
#include <stdio.h>

int fib(int n)
{
    int fm1, fm2;
    if (n < 2) {
        return n;
    } else {
        fm1 = fib(n - 1);
        fm2 = fib(n - 2);
        return fm1 + fm2;
    }
}

int main(void)
{
    int f = fib(3);
    printf("fib(3) = %d\n", f);
    return 0;
}
```



[redacted]

Rekursion (2)

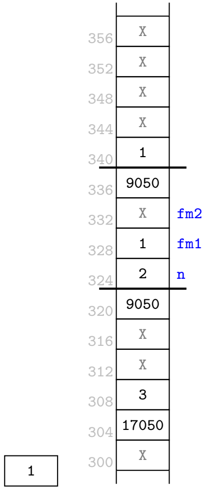
```c
#include <stdio.h>

int fib(int n)
{
    int fm1, fm2;
    if (n < 2) {
        return n;
    } else {
        fm1 = fib(n - 1);
        fm2 = fib(n - 2);
        return fm1 + fm2;
    }
}

int main(void)
{
    int f = fib(3);
    printf("fib(3) = %d\n", f);
    return 0;
}
```



[redacted]

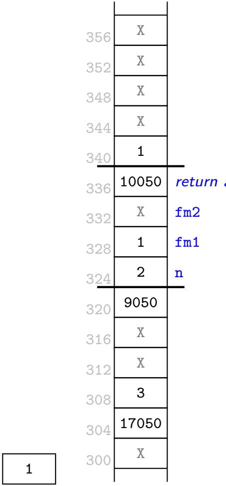
Rekursion (2)

```c
#include <stdio.h>

int fib(int n)
{
    int fm1, fm2;
    if (n < 2) {
        return n;
    } else {
        fm1 = fib(n - 1);
        fm2 = fib(n - 2);
        return fm1 + fm2;
    }
}

int main(void)
{
    int f = fib(3);
    printf("fib(3) = %d\n", f);
    return 0;
}
```



Rekursion (2)

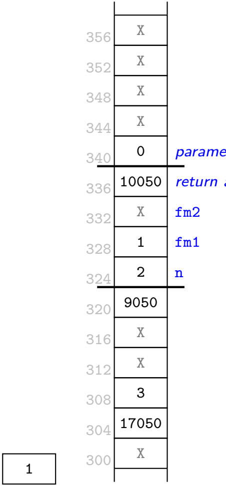
```c
#include <stdio.h>

int fib(int n)
{
    int fm1, fm2;
    if (n < 2) {
        return n;
    } else {
        fm1 = fib(n - 1);
        fm2 = fib(n - 2);
        return fm1 + fm2;
    }
}

int main(void)
{
    int f = fib(3);
    printf("fib(3) = %d\n", f);
    return 0;
}
```



[redacted]

Rekursion (2)

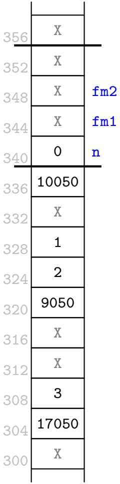
```c
#include <stdio.h>

int fib(int n)
{
    int fm1, fm2;
    if (n < 2) {
        return n;
    } else {
        fm1 = fib(n - 1);
        fm2 = fib(n - 2);
        return fm1 + fm2;
    }
}

int main(void)
{
    int f = fib(3);
    printf("fib(3) = %d\n", f);
    return 0;
}
```



[redacted]

Rekursion (2)

```c
#include <stdio.h>

int fib(int n)
{
    int fm1, fm2;
    if (n < 2) {
        return n;
    } else {
        fm1 = fib(n - 1);
        fm2 = fib(n - 2);
        return fm1 + fm2;
    }
}

int main(void)
{
    int f = fib(3);
    printf("fib(3) = %d\n", f);
    return 0;
}
```

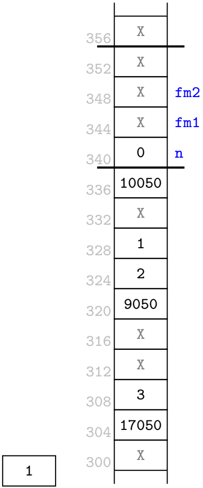


1

[redacted]

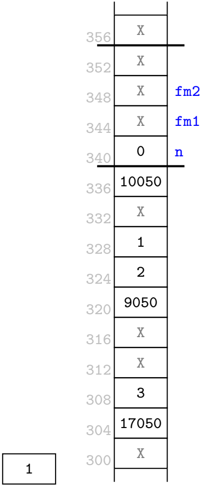
Rekursion (2)

```c
#include <stdio.h>

int fib(int n)
{
    int fm1, fm2;
    if (n < 2) {
        return n;
    } else {
        fm1 = fib(n - 1);
        fm2 = fib(n - 2);
        return fm1 + fm2;
    }
}

int main(void)
{
    int f = fib(3);
    printf("fib(3) = %d\n", f);
    return 0;
}
```



Rekursion (2)

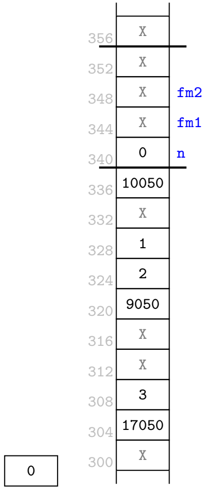
```c
#include <stdio.h>

int fib(int n)
{
    int fm1, fm2;
    if (n < 2) {
        return n;
    } else {
        fm1 = fib(n - 1);
        fm2 = fib(n - 2);
        return fm1 + fm2;
    }
}

int main(void)
{
    int f = fib(3);
    printf("fib(3) = %d\n", f);
    return 0;
}
```



[redacted]

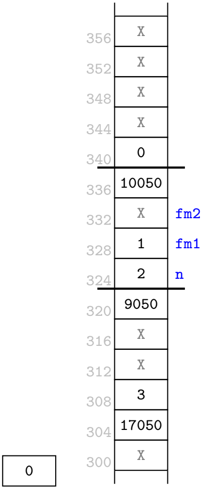
Rekursion (2)

```c
#include <stdio.h>

int fib(int n)
{
    int fm1, fm2;
    if (n < 2) {
        return n;
    } else {
        fm1 = fib(n - 1);
        fm2 = fib(n - 2);
        return fm1 + fm2;
    }
}

int main(void)
{
    int f = fib(3);
    printf("fib(3) = %d\n", f);
    return 0;
}
```



Rekursion (2)

```c
#include <stdio.h>

int fib(int n)
{
    int fm1, fm2;
    if (n < 2) {
        return n;
    } else {
        fm1 = fib(n - 1);
        fm2 = fib(n - 2);
        return fm1 + fm2;
    }
}

int main(void)
{
    int f = fib(3);
    printf("fib(3) = %d\n", f);
    return 0;
}
```



[redacted]

Rekursion (2)

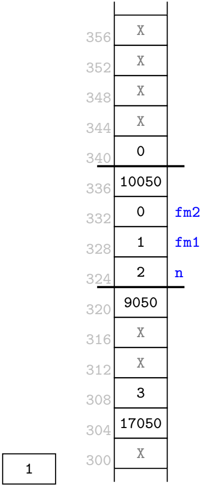
```c
#include <stdio.h>

int fib(int n)
{
    int fm1, fm2;
    if (n < 2) {
        return n;
    } else {
        fm1 = fib(n - 1);
        fm2 = fib(n - 2);
        return fm1 + fm2;
    }
}

int main(void)
{
    int f = fib(3);
    printf("fib(3) = %d\n", f);
    return 0;
}
```

Address	Value	Variable
356	X	
352	X	
348	X	
344	X	
340	0	
336	10050	
332	0	fm2
328	1	fm1
324	2	n
320	9050	
316	X	
312	X	
308	3	
304	17050	
300	X	

0

Rekursion (2)

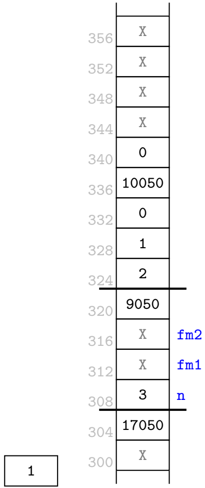
```c
#include <stdio.h>

int fib(int n)
{
    int fm1, fm2;
    if (n < 2) {
        return n;
    } else {
        fm1 = fib(n - 1);
        fm2 = fib(n - 2);
        return fm1 + fm2;
    }
}

int main(void)
{
    int f = fib(3);
    printf("fib(3) = %d\n", f);
    return 0;
}
```



[redacted]

Rekursion (2)

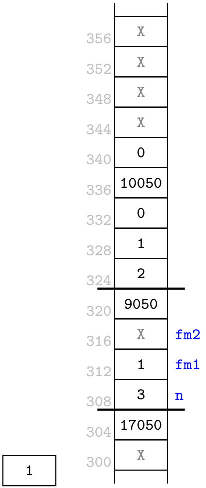
```c
#include <stdio.h>

int fib(int n)
{
    int fm1, fm2;
    if (n < 2) {
        return n;
    } else {
        fm1 = fib(n - 1);
        fm2 = fib(n - 2);
        return fm1 + fm2;
    }
}

int main(void)
{
    int f = fib(3);
    printf("fib(3) = %d\n", f);
    return 0;
}
```



[redacted]

Rekursion (2)

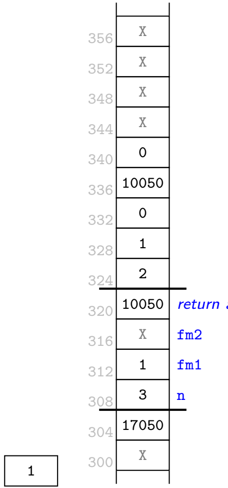
```c
#include <stdio.h>

int fib(int n)
{
    int fm1, fm2;
    if (n < 2) {
        return n;
    } else {
        fm1 = fib(n - 1);
        fm2 = fib(n - 2);
        return fm1 + fm2;
    }
}

int main(void)
{
    int f = fib(3);
    printf("fib(3) = %d\n", f);
    return 0;
}
```



[redacted]

Rekursion (2)

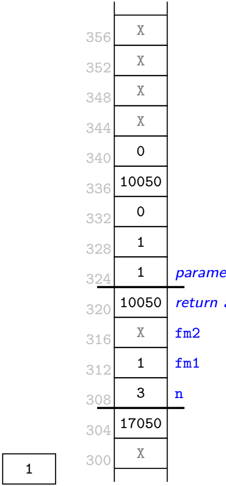
```c
#include <stdio.h>

int fib(int n)
{
    int fm1, fm2;
    if (n < 2) {
        return n;
    } else {
        fm1 = fib(n - 1);
        fm2 = fib(n - 2);
        return fm1 + fm2;
    }
}

int main(void)
{
    int f = fib(3);
    printf("fib(3) = %d\n", f);
    return 0;
}
```



[redacted]

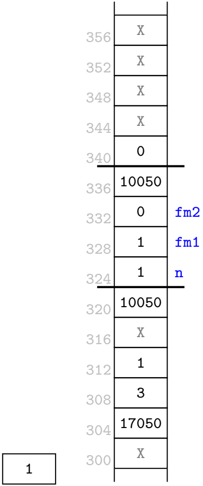
Rekursion (2)

```c
#include <stdio.h>

int fib(int n)
{
    int fm1, fm2;
    if (n < 2) {
        return n;
    } else {
        fm1 = fib(n - 1);
        fm2 = fib(n - 2);
        return fm1 + fm2;
    }
}

int main(void)
{
    int f = fib(3);
    printf("fib(3) = %d\n", f);
    return 0;
}
```



Rekursion (2)

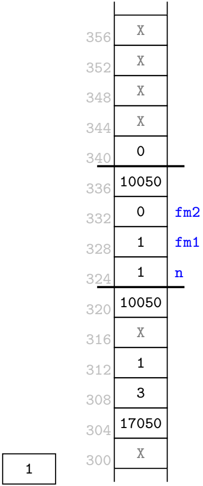
```c
#include <stdio.h>

int fib(int n)
{
    int fm1, fm2;
    if (n < 2) {
        return n;
    } else {
        fm1 = fib(n - 1);
        fm2 = fib(n - 2);
        return fm1 + fm2;
    }
}

int main(void)
{
    int f = fib(3);
    printf("fib(3) = %d\n", f);
    return 0;
}
```



[redacted]

Rekursion (2)

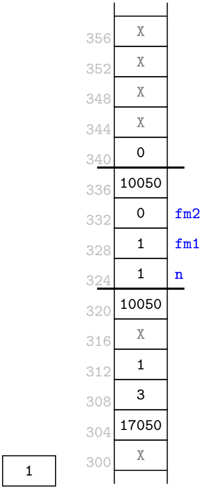
```c
#include <stdio.h>

int fib(int n)
{
    int fm1, fm2;
    if (n < 2) {
        return n;
    } else {
        fm1 = fib(n - 1);
        fm2 = fib(n - 2);
        return fm1 + fm2;
    }
}

int main(void)
{
    int f = fib(3);
    printf("fib(3) = %d\n", f);
    return 0;
}
```



[redacted]

Rekursion (2)

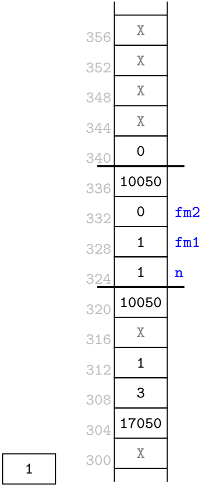
```c
#include <stdio.h>

int fib(int n)
{
    int fm1, fm2;
    if (n < 2) {
        return n;
    } else {
        fm1 = fib(n - 1);
        fm2 = fib(n - 2);
        return fm1 + fm2;
    }
}

int main(void)
{
    int f = fib(3);
    printf("fib(3) = %d\n", f);
    return 0;
}
```



[redacted]

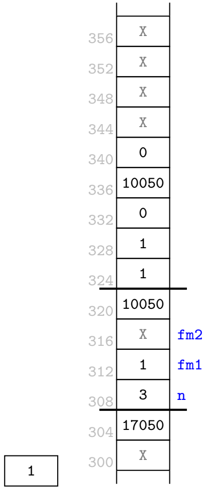
Rekursion (2)

```c
#include <stdio.h>

int fib(int n)
{
    int fm1, fm2;
    if (n < 2) {
        return n;
    } else {
        fm1 = fib(n - 1);
        fm2 = fib(n - 2);
        return fm1 + fm2;
    }
}

int main(void)
{
    int f = fib(3);
    printf("fib(3) = %d\n", f);
    return 0;
}
```



Rekursion (2)

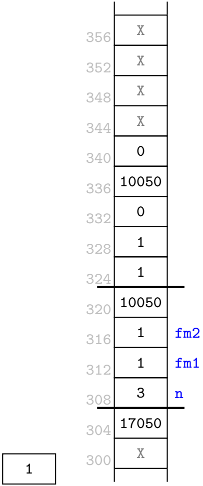
```c
#include <stdio.h>

int fib(int n)
{
    int fm1, fm2;
    if (n < 2) {
        return n;
    } else {
        fm1 = fib(n - 1);
        fm2 = fib(n - 2);
        return fm1 + fm2;
    }
}

int main(void)
{
    int f = fib(3);
    printf("fib(3) = %d\n", f);
    return 0;
}
```



[redacted]

Rekursion (2)

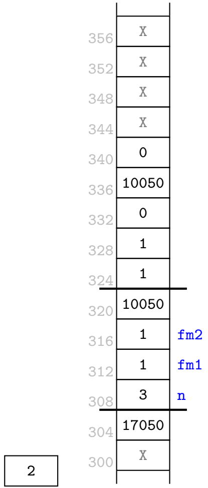
```c
#include <stdio.h>

int fib(int n)
{
    int fm1, fm2;
    if (n < 2) {
        return n;
    } else {
        fm1 = fib(n - 1);
        fm2 = fib(n - 2);
        return fm1 + fm2;
    }
}

int main(void)
{
    int f = fib(3);
    printf("fib(3) = %d\n", f);
    return 0;
}
```



[redacted]

Rekursion (2)

```c
#include <stdio.h>

int fib(int n)
{
    int fm1, fm2;
    if (n < 2) {
        return n;
    } else {
        fm1 = fib(n - 1);
        fm2 = fib(n - 2);
        return fm1 + fm2;
    }
}

int main(void)
{
    int f = fib(3);
    printf("fib(3) = %d\n", f);
    return 0;
}
```



Rekursion (2)

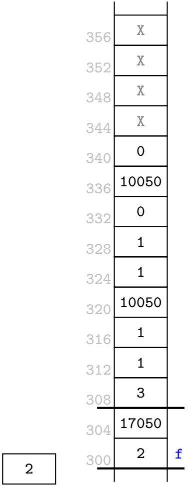
```c
#include <stdio.h>

int fib(int n)
{
    int fm1, fm2;
    if (n < 2) {
        return n;
    } else {
        fm1 = fib(n - 1);
        fm2 = fib(n - 2);
        return fm1 + fm2;
    }
}

int main(void)
{
    int f = fib(3);
    printf("fib(3) = %d\n", f);
    return 0;
}
```

356	X	
352	X	
348	X	
344	X	
340	0	
336	10050	
332	0	
328	1	
324	1	
320	10050	
316	1	
312	1	
308	3	
304	17050	
300	X	f

2

Rekursion (2)

```c
#include <stdio.h>

int fib(int n)
{
    int fm1, fm2;
    if (n < 2) {
        return n;
    } else {
        fm1 = fib(n - 1);
        fm2 = fib(n - 2);
        return fm1 + fm2;
    }
}

int main(void)
{
    int f = fib(3);
    printf("fib(3) = %d\n", f);
    return 0;
}
```




```
fib(3) = 2
```

Rekursion (2)

```c
#include <stdio.h>

int fib(int n)
{
    int fm1, fm2;
    if (n < 2) {
        return n;
    } else {
        fm1 = fib(n - 1);
        fm2 = fib(n - 2);
        return fm1 + fm2;
    }
}

int main(void)
{
    int f = fib(3);
    printf("fib(3) = %d\n", f);
    return 0;
}
```

356	X	
352	X	
348	X	
344	X	
340	0	
336	10050	
332	0	
328	1	
324	1	
320	10050	
316	1	
312	1	
308	3	
304	17050	
300	2	f

2

```
fib(3) = 2
```

Rekursion (2)

```c
#include <stdio.h>

int fib(int n)
{
    int fm1, fm2;
    if (n < 2) {
        return n;
    } else {
        fm1 = fib(n - 1);
        fm2 = fib(n - 2);
        return fm1 + fm2;
    }
}

int main(void)
{
    int f = fib(3);
    printf("fib(3) = %d\n", f);
    return 0;
}
```

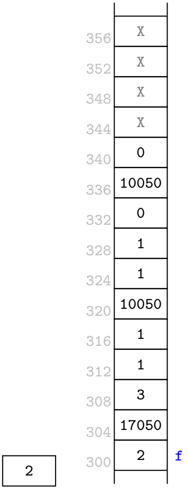
Address	Value	
356	X	
352	X	
348	X	
344	X	
340	0	
336	10050	
332	0	
328	1	
324	1	
320	10050	
316	1	
312	1	
308	3	
304	17050	
300	2	f

2

```
fib(3) = 2
```

Rekursion (2)

```c
#include <stdio.h>

int fib(int n)
{
    int fm1, fm2;
    if (n < 2) {
        return n;
    } else {
        fm1 = fib(n - 1);
        fm2 = fib(n - 2);
        return fm1 + fm2;
    }
}

int main(void)
{
    int f = fib(3);
    printf("fib(3) = %d\n", f);
    return 0;
}
```




```
fib(3) = 2
```

Rekursion (3)

code/fib.c

```c
#include <stdio.h>

int fib(int n)
{
    int fm1, fm2;
    if (n < 2) {
        return n;
    } else {
        fm1 = fib(n - 1);
        fm2 = fib(n - 2);
        return fm1 + fm2;
    }
}

int main(void)
{
    int f = fib(3);
    printf("fib(3) = %d\n", f);
    return 0;
}
```

main

Rekursion (3)

code/fib.c

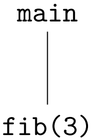
```c
#include <stdio.h>

int fib(int n)
{
    int fm1, fm2;
    if (n < 2) {
        return n;
    } else {
        fm1 = fib(n - 1);
        fm2 = fib(n - 2);
        return fm1 + fm2;
    }
}

int main(void)
{
    int f = fib(3);
    printf("fib(3) = %d\n", f);
    return 0;
}
```

main

Rekursion (3)

code/fib.c

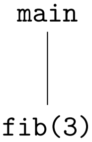
```c
#include <stdio.h>

int fib(int n)
{
    int fm1, fm2;
    if (n < 2) {
        return n;
    } else {
        fm1 = fib(n - 1);
        fm2 = fib(n - 2);
        return fm1 + fm2;
    }
}

int main(void)
{
    int f = fib(3);
    printf("fib(3) = %d\n", f);
    return 0;
}
```



Rekursion (3)

code/fib.c

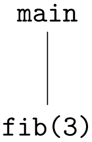
```c
#include <stdio.h>

int fib(int n)
{
    int fm1, fm2;
    if (n < 2) {
        return n;
    } else {
        fm1 = fib(n - 1);
        fm2 = fib(n - 2);
        return fm1 + fm2;
    }
}

int main(void)
{
    int f = fib(3);
    printf("fib(3) = %d\n", f);
    return 0;
}
```



Rekursion (3)

code/fib.c

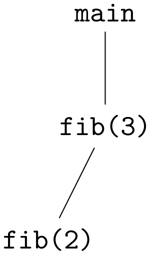
```c
#include <stdio.h>

int fib(int n)
{
    int fm1, fm2;
    if (n < 2) {
        return n;
    } else {
        fm1 = fib(n - 1);
        fm2 = fib(n - 2);
        return fm1 + fm2;
    }
}

int main(void)
{
    int f = fib(3);
    printf("fib(3) = %d\n", f);
    return 0;
}
```



Rekursion (3)

code/fib.c

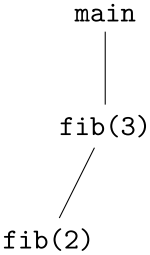
```c
#include <stdio.h>

int fib(int n)
{
    int fm1, fm2;
    if (n < 2) {
        return n;
    } else {
        fm1 = fib(n - 1);
        fm2 = fib(n - 2);
        return fm1 + fm2;
    }
}

int main(void)
{
    int f = fib(3);
    printf("fib(3) = %d\n", f);
    return 0;
}
```



Rekursion (3)

code/fib.c

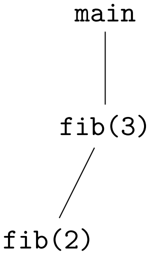
```c
#include <stdio.h>

int fib(int n)
{
    int fm1, fm2;
    if (n < 2) {
        return n;
    } else {
        fm1 = fib(n - 1);
        fm2 = fib(n - 2);
        return fm1 + fm2;
    }
}

int main(void)
{
    int f = fib(3);
    printf("fib(3) = %d\n", f);
    return 0;
}
```



Rekursion (3)

code/fib.c

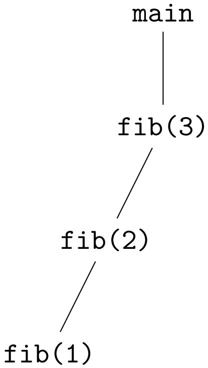
```c
#include <stdio.h>

int fib(int n)
{
    int fm1, fm2;
    if (n < 2) {
        return n;
    } else {
        fm1 = fib(n - 1);
        fm2 = fib(n - 2);
        return fm1 + fm2;
    }
}

int main(void)
{
    int f = fib(3);
    printf("fib(3) = %d\n", f);
    return 0;
}
```



Rekursion (3)

code/fib.c

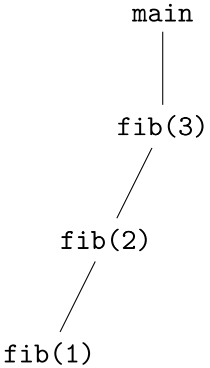
```c
#include <stdio.h>

int fib(int n)
{
    int fm1, fm2;
    if (n < 2) {
        return n;
    } else {
        fm1 = fib(n - 1);
        fm2 = fib(n - 2);
        return fm1 + fm2;
    }
}

int main(void)
{
    int f = fib(3);
    printf("fib(3) = %d\n", f);
    return 0;
}
```



Rekursion (3)

code/fib.c

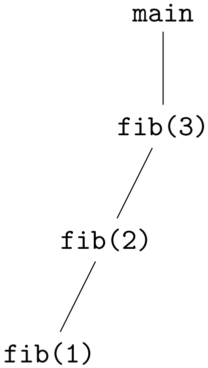
```c
#include <stdio.h>

int fib(int n)
{
    int fm1, fm2;
    if (n < 2) {
        return n;
    } else {
        fm1 = fib(n - 1);
        fm2 = fib(n - 2);
        return fm1 + fm2;
    }
}

int main(void)
{
    int f = fib(3);
    printf("fib(3) = %d\n", f);
    return 0;
}
```



Rekursion (3)

code/fib.c

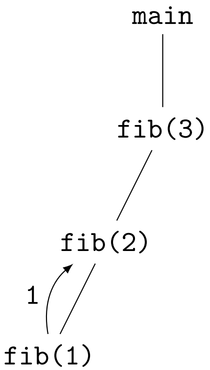
```c
#include <stdio.h>

int fib(int n)
{
    int fm1, fm2;
    if (n < 2) {
        return n;
    } else {
        fm1 = fib(n - 1);
        fm2 = fib(n - 2);
        return fm1 + fm2;
    }
}

int main(void)
{
    int f = fib(3);
    printf("fib(3) = %d\n", f);
    return 0;
}
```



Rekursion (3)

code/fib.c

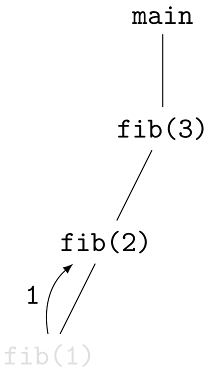
```c
#include <stdio.h>

int fib(int n)
{
    int fm1, fm2;
    if (n < 2) {
        return n;
    } else {
        fm1 = fib(n - 1);
        fm2 = fib(n - 2);
        return fm1 + fm2;
    }
}

int main(void)
{
    int f = fib(3);
    printf("fib(3) = %d\n", f);
    return 0;
}
```



Rekursion (3)

code/fib.c

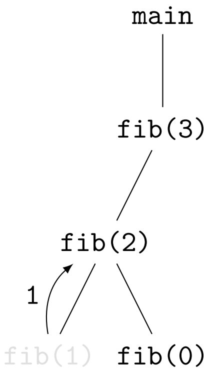
```c
#include <stdio.h>

int fib(int n)
{
    int fm1, fm2;
    if (n < 2) {
        return n;
    } else {
        fm1 = fib(n - 1);
        fm2 = fib(n - 2);
        return fm1 + fm2;
    }
}

int main(void)
{
    int f = fib(3);
    printf("fib(3) = %d\n", f);
    return 0;
}
```



Rekursion (3)

code/fib.c

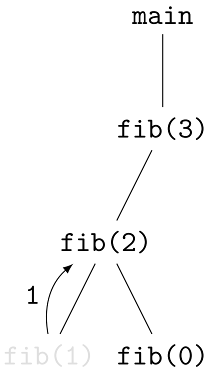
```c
#include <stdio.h>

int fib(int n)
{
    int fm1, fm2;
    if (n < 2) {
        return n;
    } else {
        fm1 = fib(n - 1);
        fm2 = fib(n - 2);
        return fm1 + fm2;
    }
}

int main(void)
{
    int f = fib(3);
    printf("fib(3) = %d\n", f);
    return 0;
}
```



Rekursion (3)

code/fib.c

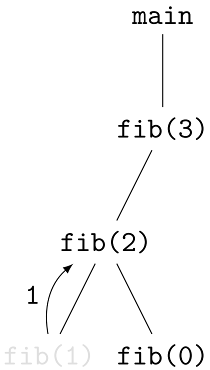
```c
#include <stdio.h>

int fib(int n)
{
    int fm1, fm2;
    if (n < 2) {
        return n;
    } else {
        fm1 = fib(n - 1);
        fm2 = fib(n - 2);
        return fm1 + fm2;
    }
}

int main(void)
{
    int f = fib(3);
    printf("fib(3) = %d\n", f);
    return 0;
}
```



Rekursion (3)

code/fib.c

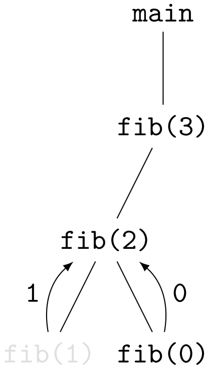
```c
#include <stdio.h>

int fib(int n)
{
    int fm1, fm2;
    if (n < 2) {
        return n;
    } else {
        fm1 = fib(n - 1);
        fm2 = fib(n - 2);
        return fm1 + fm2;
    }
}

int main(void)
{
    int f = fib(3);
    printf("fib(3) = %d\n", f);
    return 0;
}
```



Rekursion (3)

code/fib.c

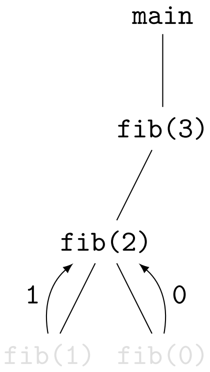
```c
#include <stdio.h>

int fib(int n)
{
    int fm1, fm2;
    if (n < 2) {
        return n;
    } else {
        fm1 = fib(n - 1);
        fm2 = fib(n - 2);
        return fm1 + fm2;
    }
}

int main(void)
{
    int f = fib(3);
    printf("fib(3) = %d\n", f);
    return 0;
}
```



Rekursion (3)

code/fib.c

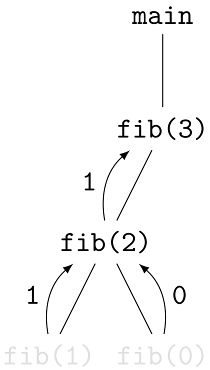
```c
#include <stdio.h>

int fib(int n)
{
    int fm1, fm2;
    if (n < 2) {
        return n;
    } else {
        fm1 = fib(n - 1);
        fm2 = fib(n - 2);
        return fm1 + fm2;
    }
}

int main(void)
{
    int f = fib(3);
    printf("fib(3) = %d\n", f);
    return 0;
}
```



Rekursion (3)

code/fib.c

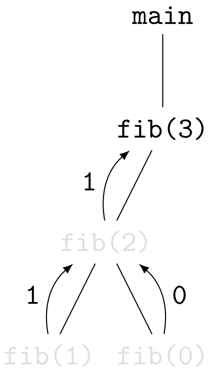
```c
#include <stdio.h>

int fib(int n)
{
    int fm1, fm2;
    if (n < 2) {
        return n;
    } else {
        fm1 = fib(n - 1);
        fm2 = fib(n - 2);
        return fm1 + fm2;
    }
}

int main(void)
{
    int f = fib(3);
    printf("fib(3) = %d\n", f);
    return 0;
}
```



Rekursion (3)

code/fib.c

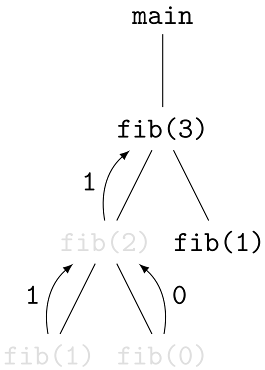
```c
#include <stdio.h>

int fib(int n)
{
    int fm1, fm2;
    if (n < 2) {
        return n;
    } else {
        fm1 = fib(n - 1);
        fm2 = fib(n - 2);
        return fm1 + fm2;
    }
}

int main(void)
{
    int f = fib(3);
    printf("fib(3) = %d\n", f);
    return 0;
}
```



Rekursion (3)

code/fib.c

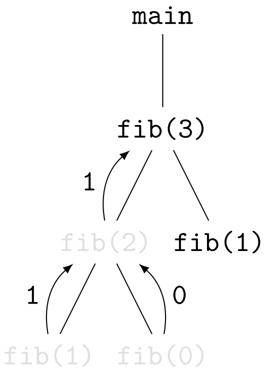
```c
#include <stdio.h>

int fib(int n)
{
    int fm1, fm2;
    if (n < 2) {
        return n;
    } else {
        fm1 = fib(n - 1);
        fm2 = fib(n - 2);
        return fm1 + fm2;
    }
}

int main(void)
{
    int f = fib(3);
    printf("fib(3) = %d\n", f);
    return 0;
}
```



Rekursion (3)

code/fib.c

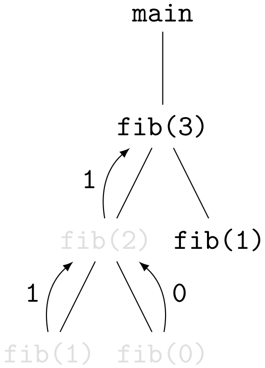
```c
#include <stdio.h>

int fib(int n)
{
    int fm1, fm2;
    if (n < 2) {
        return n;
    } else {
        fm1 = fib(n - 1);
        fm2 = fib(n - 2);
        return fm1 + fm2;
    }
}

int main(void)
{
    int f = fib(3);
    printf("fib(3) = %d\n", f);
    return 0;
}
```



Rekursion (3)

code/fib.c

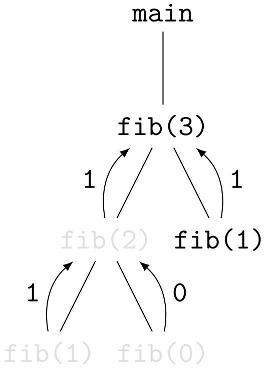
```c
#include <stdio.h>

int fib(int n)
{
    int fm1, fm2;
    if (n < 2) {
        return n;
    } else {
        fm1 = fib(n - 1);
        fm2 = fib(n - 2);
        return fm1 + fm2;
    }
}

int main(void)
{
    int f = fib(3);
    printf("fib(3) = %d\n", f);
    return 0;
}
```



Rekursion (3)

code/fib.c

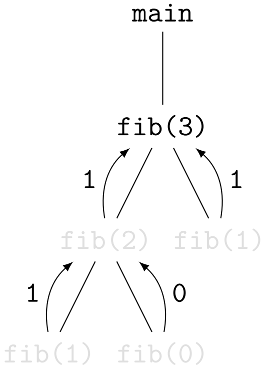
```c
#include <stdio.h>

int fib(int n)
{
    int fm1, fm2;
    if (n < 2) {
        return n;
    } else {
        fm1 = fib(n - 1);
        fm2 = fib(n - 2);
        return fm1 + fm2;
    }
}

int main(void)
{
    int f = fib(3);
    printf("fib(3) = %d\n", f);
    return 0;
}
```



Rekursion (3)

code/fib.c

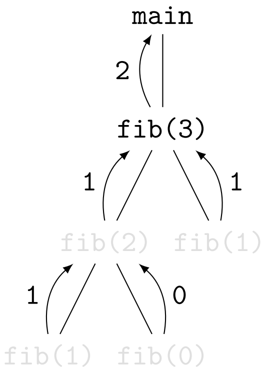
```c
#include <stdio.h>

int fib(int n)
{
    int fm1, fm2;
    if (n < 2) {
        return n;
    } else {
        fm1 = fib(n - 1);
        fm2 = fib(n - 2);
        return fm1 + fm2;
    }
}

int main(void)
{
    int f = fib(3);
    printf("fib(3) = %d\n", f);
    return 0;
}
```



Rekursion (3)

code/fib.c

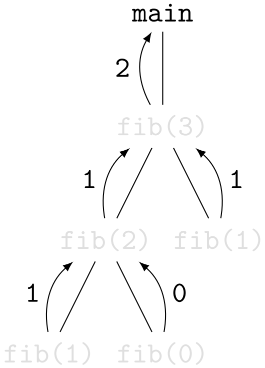
```c
#include <stdio.h>

int fib(int n)
{
    int fm1, fm2;
    if (n < 2) {
        return n;
    } else {
        fm1 = fib(n - 1);
        fm2 = fib(n - 2);
        return fm1 + fm2;
    }
}

int main(void)
{
    int f = fib(3);
    printf("fib(3) = %d\n", f);
    return 0;
}
```



Rekursion (3)

code/fib.c

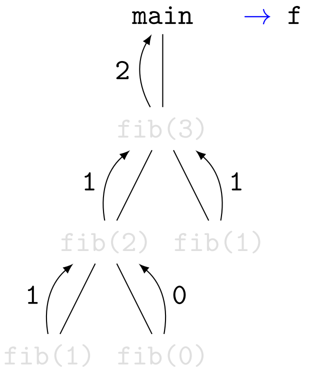
```c
#include <stdio.h>

int fib(int n)
{
    int fm1, fm2;
    if (n < 2) {
        return n;
    } else {
        fm1 = fib(n - 1);
        fm2 = fib(n - 2);
        return fm1 + fm2;
    }
}

int main(void)
{
    int f = fib(3);
    printf("fib(3) = %d\n", f);
    return 0;
}
```



Rekursion (3)

code/fib.c

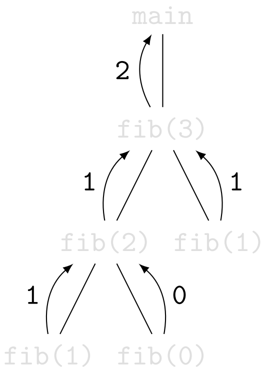
```c
#include <stdio.h>

int fib(int n)
{
    int fm1, fm2;
    if (n < 2) {
        return n;
    } else {
        fm1 = fib(n - 1);
        fm2 = fib(n - 2);
        return fm1 + fm2;
    }
}

int main(void)
{
    int f = fib(3);
    printf("fib(3) = %d\n", f);
    return 0;
}
```



Rekursion (3)

code/fib.c

```c
#include <stdio.h>

int fib(int n)
{
    int fm1, fm2;
    if (n < 2) {
        return n;
    } else {
        fm1 = fib(n - 1);
        fm2 = fib(n - 2);
        return fm1 + fm2;
    }
}

int main(void)
{
    int f = fib(3);
    printf("fib(3) = %d\n", f);
    return 0;
}
```

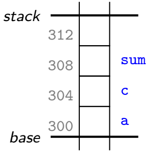


Blank

Adresser och pekare

Adress-operatorn

- Alla variabler ligger nånstans i minnet
- Alla har en adress
- Adress-operatorn & returnerar *adressen* till variabeln
  - *ej* värdet



code/vars-on-stack.c

```c
int main(void)
{
    int a = 2;
    int c = 3;
    int sum;
```

- `printf("The address of c=%p\n", &c);`
  - `The address of c=304`

Pekare (1)

- En pekare eller pekarvariabel är en variabel som innehåller en adress
  - Adressen anger var i minnet variabeln börjar
- Internt lagras den som ett heltal
  - 32 eller 64 bitar (4 eller 8 bytes) beroende på system
  - Detta dokument använder 4 bytes
- En pekare deklareras med en stjärna (∗) efter typen
  - Typen kan vara en enkel typ, inklusive pekartyper eller en **`struct`**

Pekare (2)

- Till exempel:
  - ```
    int *p;
    struct cell *q;
    char *r;
    ```
  - Variabeln `p` är av typen "pekare till `int`" (*"`int` pointer"* eller *"pointer to `int`"*)
  - Variabeln `q` är av typen "pekare till `struct cell`" (*"`struct cell` pointer"* eller *"pointer to `struct cell`"*)
  - Variabeln `r` är av typen "pekare till `char`" (*"`char` pointer"* eller *"pointer to `char`"*)
    ```

Pekare (3)

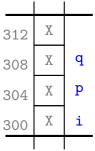
- Inget hindrar att vi har en pekare till en pekare
  - `int **q;`
  - Här är variabeln q av typen "pekare till pekare till **int**" eller "dubbelpekare till **int**" (*"int double pointer"*)
- Notera att om flera variabler deklareras i samma sats så är stjärnan kopplad till variabeln, inte till typen
  - Exempel:
    - `int i, *p, **q;`

    deklarerar
    - en variabel `i` av typen `int`
    - en variabel `p` av typen `int *` (enkelpekare)
    - en variabel `q` av typen `int **` (dubbelpekare)

Pekare och adress-operatorn

- En pekare till typen X kan tilldelas adressen för en variabel av typen X

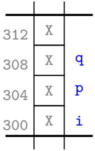
```c
int i, *p, **q;
p = &i;
q = &p;
```



Pekare och adress-operatorn

- En pekare till typen X kan tilldelas adressen för en variabel av typen X

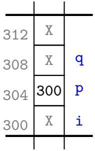
```
int i, *p, **q;
p = &i;
q = &p;
```



Pekare och adress-operatorn

- En pekare till typen X kan tilldelas adressen för en variabel av typen X

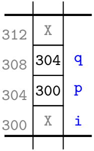
```c
int i, *p, **q;
p = &i;
q = &p;
```



Pekare och adress-operatorn

- En pekare till typen X kan tilldelas adressen för en variabel av typen X

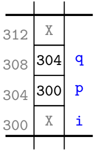
```c
int i, *p, **q;
p = &i;
q = &p;
```



Pekare och adress-operatorn

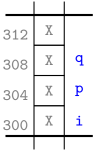
- En pekare till typen X kan tilldelas adressen för en variabel av typen X

```c
int i, *p, **q;
p = &i;
q = &p;
```



- Ofta illustrerar man pekare med hjälp av en pil

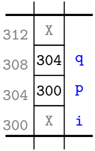
```c
int i, *p, **q;
p = &i;
q = &p;
```



Pekare och adress-operatorn

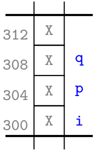
- En pekare till typen X kan tilldelas adressen för en variabel av typen X

```c
int i, *p, **q;
p = &i;
q = &p;
```



- Ofta illustrerar man pekare med hjälp av en pil

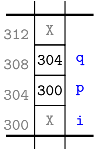
```c
int i, *p, **q;
p = &i;
q = &p;
```



Pekare och adress-operatorn

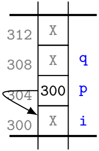
- En pekare till typen X kan tilldelas adressen för en variabel av typen X

```c
int i, *p, **q;
p = &i;
q = &p;
```



- Ofta illustrerar man pekare med hjälp av en pil

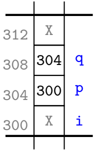
```c
int i, *p, **q;
p = &i;
q = &p;
```



Pekare och adress-operatorn

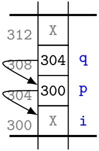
- En pekare till typen X kan tilldelas adressen för en variabel av typen X

```
int i, *p, **q;
p = &i;
q = &p;
```



- Ofta illustrerar man pekare med hjälp av en pil

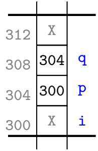
```
int i, *p, **q;
p = &i;
q = &p;
```



Pekare och adress-operatorn

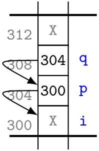
- En pekare till typen X kan tilldelas adressen för en variabel av typen X

```c
int i, *p, **q;
p = &i;
q = &p;
```



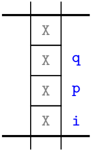
- Ofta illustrerar man pekare med hjälp av en pil

```c
int i, *p, **q;
p = &i;
q = &p;
```



- Oftast utelämnar man adresserna helt

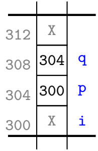
```c
int i, *p, **q;
p = &i;
q = &p;
```



Pekare och adress-operatorn

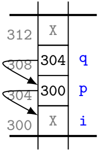
- En pekare till typen X kan tilldelas adressen för en variabel av typen X

```c
int i, *p, **q;
p = &i;
q = &p;
```



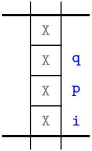
- Ofta illustrerar man pekare med hjälp av en pil

```c
int i, *p, **q;
p = &i;
q = &p;
```



- Oftast utelämnar man adresserna helt

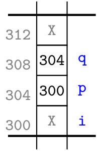
```c
int i, *p, **q;
p = &i;
q = &p;
```



Pekare och adress-operatorn

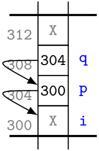
- En pekare till typen X kan tilldelas adressen för en variabel av typen X

```c
int i, *p, **q;
p = &i;
q = &p;
```



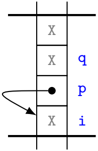
- Ofta illustrerar man pekare med hjälp av en pil

```c
int i, *p, **q;
p = &i;
q = &p;
```



- Oftast utelämnar man adresserna helt

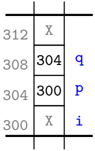
```c
int i, *p, **q;
p = &i;
q = &p;
```



Pekare och adress-operatorn

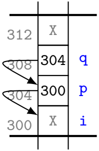
- En pekare till typen X kan tilldelas adressen för en variabel av typen X

```c
int i, *p, **q;
p = &i;
q = &p;
```



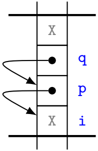
- Ofta illustrerar man pekare med hjälp av en pil

```c
int i, *p, **q;
p = &i;
q = &p;
```



- Oftast utelämnar man adresserna helt

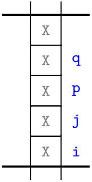
```c
int i, *p, **q;
p = &i;
q = &p;
```



Dereferering

- Studera koden

```
int i, j, *p, *q;
p = &i;
q = &i;
*p = 4; // Same effect as i=4
j = *q; // Same effect as j=i
```

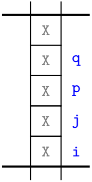


- Om p pekar på (refererar till) variabeln i så kan vi dereferera p för att komma åt värdet i i
  - Det kallas ibland att vi följer pekaren
- Det görs genom att skriva en stjärna framför variabelnamnet
  - Om p är av typen `int *` så är uttrycket `*p` av typen `int`
- Att p och q pekar på samma variabel kallas för aliasing
  - Aliasing behövs ofta, men öppnar för buggar

Dereferering

- Studera koden

```c
int i, j, *p, *q;
p = &i;
q = &i;
*p = 4; // Same effect as i=4
j = *q; // Same effect as j=i
```

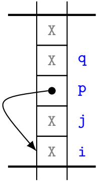


- Om `p` pekar på (**refererar till**) variabeln `i` så kan vi **dereferera** `p` för att komma åt **värdet** i `i`
  - Det kallas ibland att vi **följer** pekaren
- Det görs genom att skriva en stjärna **framför** variabelnamnet
  - Om `p` är av typen `int *` så är uttrycket `*p` av typen `int`
- Att `p` och `q` pekar på samma variabel kallas för **aliasing**
  - Aliasing behövs ofta, men öppnar för buggar

Dereferering

- Studera koden

```c
int i, j, *p, *q;
p = &i;
q = &i;
*p = 4; // Same effect as i=4
j = *q; // Same effect as j=i
```

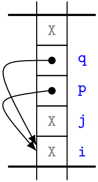


- Om p pekar på (refererar till) variabeln `i` så kan vi dereferera `p` för att komma åt värdet i `i`
  - Det kallas ibland att vi följer pekaren
- Det görs genom att skriva en stjärna framför variabelnamnet
  - Om `p` är av typen `int *` så är uttrycket `*p` av typen `int`
- Att `p` och `q` pekar på samma variabel kallas för aliasing
  - Aliasing behövs ofta, men öppnar för buggar

Dereferering

- Studera koden

```
int i, j, *p, *q;
p = &i;
q = &i;
*p = 4; // Same effect as i=4
j = *q; // Same effect as j=i
```

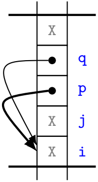


- Om p pekar på (refererar till) variabeln i så kan vi dereferera p för att komma åt värdet i i
  - Det kallas ibland att vi följer pekaren
- Det görs genom att skriva en stjärna framför variabelnamnet
  - Om p är av typen `int *` så är uttrycket `*p` av typen `int`
- Att p och q pekar på samma variabel kallas för aliasing
  - Aliasing behövs ofta, men öppnar för buggar

Dereferering

- Studera koden

```c
int i, j, *p, *q;
p = &i;
q = &i;
*p = 4; // Same effect as i=4
j = *q; // Same effect as j=i
```

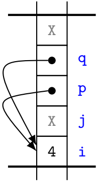


- Om p pekar på (refererar till) variabeln i så kan vi dereferera p för att komma åt värdet i i
  - Det kallas ibland att vi följer pekaren
- Det görs genom att skriva en stjärna framför variabelnamnet
  - Om p är av typen `int *` så är uttrycket `*p` av typen `int`
- Att p och q pekar på samma variabel kallas för aliasing
  - Aliasing behövs ofta, men öppnar för buggar

Dereferering

- Studera koden

```c
int i, j, *p, *q;
p = &i;
q = &i;
*p = 4; // Same effect as i=4
j = *q; // Same effect as j=i
```

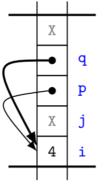


- Om p pekar på (refererar till) variabeln i så kan vi dereferera p för att komma åt värdet i i
  - Det kallas ibland att vi följer pekaren
- Det görs genom att skriva en stjärna framför variabelnamnet
  - Om p är av typen `int *` så är uttrycket `*p` av typen `int`
- Att p och q pekar på samma variabel kallas för aliasing
  - Aliasing behövs ofta, men öppnar för buggar

Dereferering

- Studera koden

```
int i, j, *p, *q;
p = &i;
q = &i;
*p = 4; // Same effect as i=4
j = *q; // Same effect as j=i
```

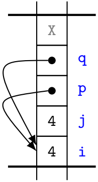


- Om p pekar på (refererar till) variabeln i så kan vi dereferera p för att komma åt värdet i i
  - Det kallas ibland att vi följer pekaren
- Det görs genom att skriva en stjärna framför variabelnamnet
  - Om p är av typen `int *` så är uttrycket `*p` av typen `int`
- Att p och q pekar på samma variabel kallas för aliasing
  - Aliasing behövs ofta, men öppnar för buggar

Dereferering

- Studera koden

```
int i, j, *p, *q;
p = &i;
q = &i;
*p = 4; // Same effect as i=4
j = *q; // Same effect as j=i
```



- Om p pekar på (refererar till) variabeln i så kan vi dereferera p för att komma åt värdet i i
  - Det kallas ibland att vi följer pekaren
- Det görs genom att skriva en stjärna framför variabelnamnet
  - Om p är av typen `int *` så är uttrycket `*p` av typen `int`
- Att p och q pekar på samma variabel kallas för aliasing
  - Aliasing behövs ofta, men öppnar för buggar

Pekare som parametrar (1)

- Vad kommer att skrivas ut av den här koden?

code/add-one1.c

```c
#include <stdio.h>
void add_one(int n)
{
    n = n + 1;
}
int main(void)
{
    int a = 5;
    add_one(a);
    printf("a = %d\n", a);
    return 0;
}
```

- ...och den här?

code/add-one2.c

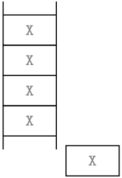
```c
#include <stdio.h>
void add_one(int *n)
{
    *n = *n + 1;
}
int main(void)
{
    int a = 5;
    add_one(&a);
    printf("a = %d\n", a);
    return 0;
}
```

Pekare som parametrar (2)

- Vi testkör!

code/add-one1.c

```c
#include <stdio.h>
void add_one(int n)
{
    n = n + 1;
}
int main(void)
{
    int a = 5;
    add_one(a);
    printf("a = %d\n", a);
    return 0;
}
```



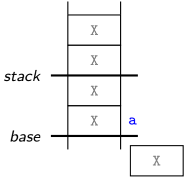
[redacted]

Pekare som parametrar (2)

- Vi testkör!

code/add-one1.c

```c
#include <stdio.h>
void add_one(int n)
{
    n = n + 1;
}
int main(void)
{
    int a = 5;
    add_one(a);
    printf("a = %d\n", a);
    return 0;
}
```

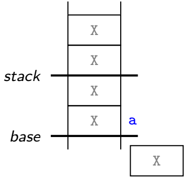


Pekare som parametrar (2)

- Vi testkör!

code/add-one1.c

```c
#include <stdio.h>
void add_one(int n)
{
    n = n + 1;
}
int main(void)
{
    int a = 5;
    add_one(a);
    printf("a = %d\n", a);
    return 0;
}
```

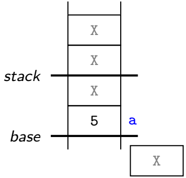


Pekare som parametrar (2)

- Vi testkör!

code/add-one1.c

```c
#include <stdio.h>
void add_one(int n)
{
    n = n + 1;
}
int main(void)
{
    int a = 5;
    add_one(a);
    printf("a = %d\n", a);
    return 0;
}
```



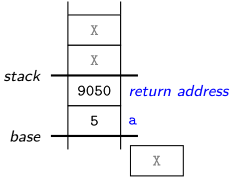
[redacted]

Pekare som parametrar (2)

- Vi testkör!

code/add-one1.c

```c
#include <stdio.h>
void add_one(int n)
{
    n = n + 1;
}
int main(void)
{
    int a = 5;
    add_one(a);
    printf("a = %d\n", a);
    return 0;
}
```

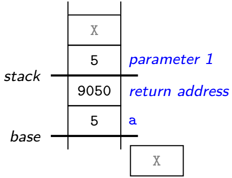


Pekare som parametrar (2)

- Vi testkör!

code/add-one1.c

```c
#include <stdio.h>
void add_one(int n)
{
    n = n + 1;
}
int main(void)
{
    int a = 5;
    add_one(a);
    printf("a = %d\n", a);
    return 0;
}
```



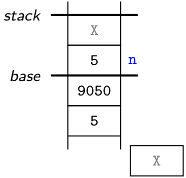
[redacted]

Pekare som parametrar (2)

- Vi testkör!

code/add-one1.c

```c
#include <stdio.h>
void add_one(int n)
{
    n = n + 1;
}
int main(void)
{
    int a = 5;
    add_one(a);
    printf("a = %d\n", a);
    return 0;
}
```

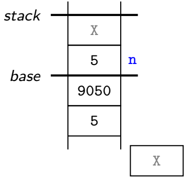


Pekare som parametrar (2)

- Vi testkör!

code/add-one1.c

```c
#include <stdio.h>
void add_one(int n)
{
    n = n + 1;
}
int main(void)
{
    int a = 5;
    add_one(a);
    printf("a = %d\n", a);
    return 0;
}
```

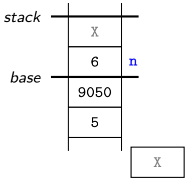


Pekare som parametrar (2)

- Vi testkör!

code/add-one1.c

```c
#include <stdio.h>
void add_one(int n)
{
    n = n + 1;
}
int main(void)
{
    int a = 5;
    add_one(a);
    printf("a = %d\n", a);
    return 0;
}
```



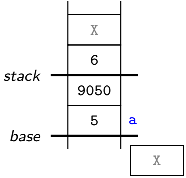
[redacted]

Pekare som parametrar (2)

- Vi testkör!

code/add-one1.c

```c
#include <stdio.h>
void add_one(int n)
{
    n = n + 1;
}
int main(void)
{
    int a = 5;
    add_one(a);
    printf("a = %d\n", a);
    return 0;
}
```



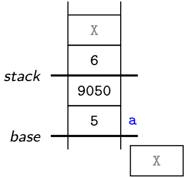
[redacted]

Pekare som parametrar (2)

- Vi testkör!

code/add-one1.c

```c
#include <stdio.h>
void add_one(int n)
{
    n = n + 1;
}
int main(void)
{
    int a = 5;
    add_one(a);
    printf("a = %d\n", a);
    return 0;
}
```



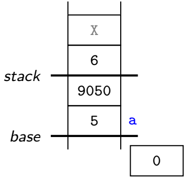

```
a = 5
```

Pekare som parametrar (2)

- Vi testkör!

code/add-one1.c

```c
#include <stdio.h>
void add_one(int n)
{
    n = n + 1;
}
int main(void)
{
    int a = 5;
    add_one(a);
    printf("a = %d\n", a);
    return 0;
}
```



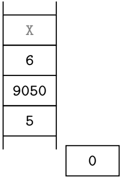

```
a = 5
```

Pekare som parametrar (2)

- Vi testkör!

code/add-one1.c

```c
#include <stdio.h>
void add_one(int n)
{
    n = n + 1;
}
int main(void)
{
    int a = 5;
    add_one(a);
    printf("a = %d\n", a);
    return 0;
}
```



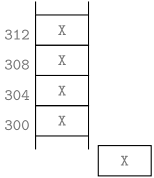

```
a = 5
```

Pekare som parametrar (3)

- Vi testkör det andra examplet!

code/add-one2.c

```c
#include <stdio.h>
void add_one(int *n)
{
    *n = *n + 1;
}
int main(void)
{
    int a = 5;
    add_one(&a);
    printf("a = %d\n", a);
    return 0;
}
```



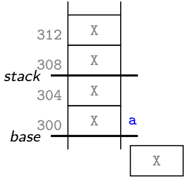
[redacted]

Pekare som parametrar (3)

- Vi testkör det andra examplet!

code/add-one2.c

```c
#include <stdio.h>
void add_one(int *n)
{
    *n = *n + 1;
}
int main(void)
{
    int a = 5;
    add_one(&a);
    printf("a = %d\n", a);
    return 0;
}
```

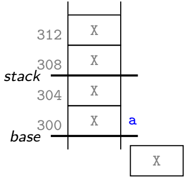


Pekare som parametrar (3)

- Vi testkör det andra examplet!

code/add-one2.c

```c
#include <stdio.h>
void add_one(int *n)
{
    *n = *n + 1;
}
int main(void)
{
    int a = 5;
    add_one(&a);
    printf("a = %d\n", a);
    return 0;
}
```

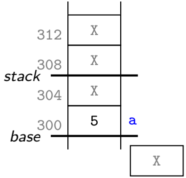


Pekare som parametrar (3)

- Vi testkör det andra examplet!

code/add-one2.c

```c
#include <stdio.h>
void add_one(int *n)
{
    *n = *n + 1;
}
int main(void)
{
    int a = 5;
    add_one(&a);
    printf("a = %d\n", a);
    return 0;
}
```

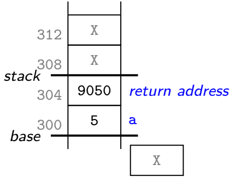


Pekare som parametrar (3)

- Vi testkör det andra examplet!

code/add-one2.c

```c
#include <stdio.h>
void add_one(int *n)
{
    *n = *n + 1;
}
int main(void)
{
    int a = 5;
    add_one(&a);
    printf("a = %d\n", a);
    return 0;
}
```

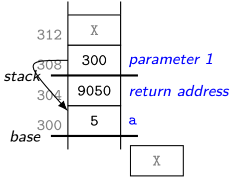


Pekare som parametrar (3)

- Vi testkör det andra examplet!

code/add-one2.c

```c
#include <stdio.h>
void add_one(int *n)
{
    *n = *n + 1;
}
int main(void)
{
    int a = 5;
    add_one(&a);
    printf("a = %d\n", a);
    return 0;
}
```

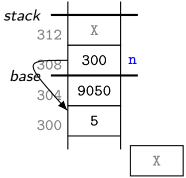


Pekare som parametrar (3)

- Vi testkör det andra examplet!

code/add-one2.c

```c
#include <stdio.h>
void add_one(int *n)
{
    *n = *n + 1;
}
int main(void)
{
    int a = 5;
    add_one(&a);
    printf("a = %d\n", a);
    return 0;
}
```

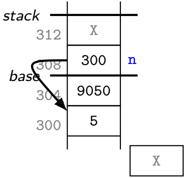


Pekare som parametrar (3)

- Vi testkör det andra examplet!

code/add-one2.c

```c
#include <stdio.h>
void add_one(int *n)
{
    *n = *n + 1;
}
int main(void)
{
    int a = 5;
    add_one(&a);
    printf("a = %d\n", a);
    return 0;
}
```

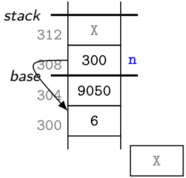


Pekare som parametrar (3)

- Vi testkör det andra examplet!

code/add-one2.c

```
#include <stdio.h>
void add_one(int *n)
{
    *n = *n + 1;
}
int main(void)
{
    int a = 5;
    add_one(&a);
    printf("a = %d\n", a);
    return 0;
}
```

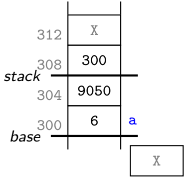


Pekare som parametrar (3)

- Vi testkör det andra examplet!

code/add-one2.c

```
#include <stdio.h>
void add_one(int *n)
{
    *n = *n + 1;
}
int main(void)
{
    int a = 5;
    add_one(&a);
    printf("a = %d\n", a);
    return 0;
}
```



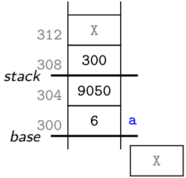
[redacted]

Pekare som parametrar (3)

- Vi testkör det andra examplet!

code/add-one2.c

```c
#include <stdio.h>
void add_one(int *n)
{
    *n = *n + 1;
}
int main(void)
{
    int a = 5;
    add_one(&a);
    printf("a = %d\n", a);
    return 0;
}
```



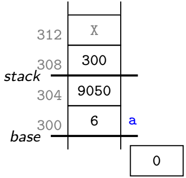

```
a = 6
```

Pekare som parametrar (3)

- Vi testkör det andra examplet!

code/add-one2.c

```
#include <stdio.h>
void add_one(int *n)
{
    *n = *n + 1;
}
int main(void)
{
    int a = 5;
    add_one(&a);
    printf("a = %d\n", a);
    return 0;
}
```



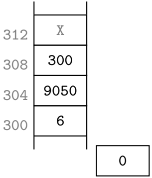

```
a = 6
```

Pekare som parametrar (3)

- Vi testkör det andra examplet!

code/add-one2.c

```c
#include <stdio.h>
void add_one(int *n)
{
    *n = *n + 1;
}
int main(void)
{
    int a = 5;
    add_one(&a);
    printf("a = %d\n", a);
    return 0;
}
```




```
a = 6
```

Vad är skillnaden?

- När funktionen tar emot en `int`-variabel så skickas en kopia av värdet som är lagrat i variabeln `a`
  - Funktionen kan inte ändra värdet som lagras i `a`
- När funktionen tar emot en pekare och vi skickar adressen till variabeln `a` kan funktionen ändra vad som finns lagrat i `a` via pekaren
- Via pekare kan en funktion ändra variabler utanför sin *stack frame*
  - i princip var som helst i minnet...

Returvärden från funktioner (1)

- Det normala sättet att returnera värden från en funktion är med `return`
  - Fungerar för enkla datatyper, t.ex. `int`, `double`
  - Fungerar för endast ett returvärde
  - Exempel: `sin(x)`
- Pekarparametrar gör det möjligt att ”returnera” flera värden
  - Exempel:

```c
void swap(int *v1, int *v2)
{
    int d = *v1;
    *v1 = *v2;
    *v2 = d;
}
```

  - Här fungerar pekarparametrarna `v1` och `v2` både som in-parametrar och ut-parametrar

Returvärden från funktioner (2)

- En del funktioner kombinerar pekarparametrar med returvärden
- Då används ofta returvärdet som en signal om allt gick som det skulle med de övriga parametrarna
- Exempel:
  - Funktionen `scanf()` tar pekare för att lagra värden
  - Returvärdet från `scanf()` är i normalfallet antalet lyckade matchningar
  - Testet

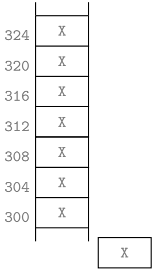
    ```
    if (scanf("%d,%d", &a, &b) == 2) {
        // We have good values in a and b
    } else {
        // Do error handling
    }
    ```

    kan användas för att säkerställa att vi bara jobbar på giltiga värden för `a` och `b`

Ett till exempel

code/swap.c

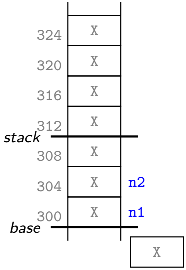
```c
#include <stdio.h>
void swap(int *v1, int *v2)
{
    int tmp;
    tmp = *v1;
    *v1 = *v2;
    *v2 = tmp;
}
int main(void)
{
    int n1 = 3, n2 = 418;
    swap(&n1, &n2);
    printf("n1 = %4.2f, n2 = %4.2f\n", n1, n2);
    return 0;
}
```



Ett till exempel

code/swap.c

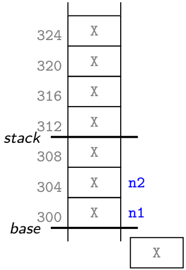
```c
#include <stdio.h>
void swap(int *v1, int *v2)
{
    int tmp;
    tmp = *v1;
    *v1 = *v2;
    *v2 = tmp;
}
int main(void)
{
    int n1 = 3, n2 = 418;
    swap(&n1, &n2);
    printf("n1 = %4.2f, n2 = %4.2f\n", n1, n2);
    return 0;
}
```



Ett till exempel

code/swap.c

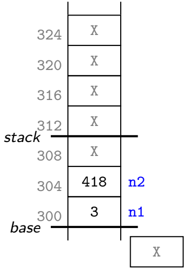
```
#include <stdio.h>
void swap(int *v1, int *v2)
{
    int tmp;
    tmp = *v1;
    *v1 = *v2;
    *v2 = tmp;
}
int main(void)
{
    int n1 = 3, n2 = 418;
    swap(&n1, &n2);
    printf("n1 = %4.2f, n2 = %4.2f\n", n1, n2);
    return 0;
}
```



Ett till exempel

code/swap.c

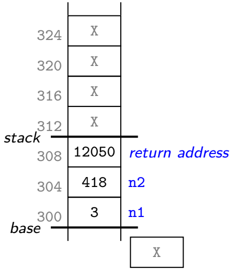
```c
#include <stdio.h>
void swap(int *v1, int *v2)
{
    int tmp;
    tmp = *v1;
    *v1 = *v2;
    *v2 = tmp;
}
int main(void)
{
    int n1 = 3, n2 = 418;
    swap(&n1, &n2);
    printf("n1 = %4.2f, n2 = %4.2f\n", n1, n2);
    return 0;
}
```



Ett till exempel

code/swap.c

```c
#include <stdio.h>
void swap(int *v1, int *v2)
{
    int tmp;
    tmp = *v1;
    *v1 = *v2;
    *v2 = tmp;
}
int main(void)
{
    int n1 = 3, n2 = 418;
    swap(&n1, &n2);
    printf("n1 = %4.2f, n2 = %4.2f\n", n1, n2);
    return 0;
}
```

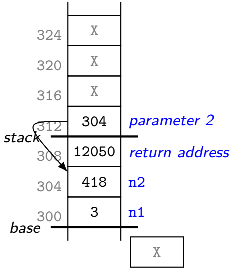


[redacted]

Ett till exempel

code/swap.c

```c
#include <stdio.h>
void swap(int *v1, int *v2)
{
    int tmp;
    tmp = *v1;
    *v1 = *v2;
    *v2 = tmp;
}
int main(void)
{
    int n1 = 3, n2 = 418;
    swap(&n1, &n2);
    printf("n1 = %4.2f, n2 = %4.2f\n", n1, n2);
    return 0;
}
```

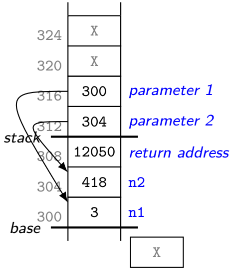


[redacted]

Ett till exempel

code/swap.c

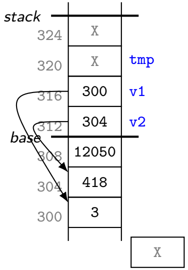
```c
#include <stdio.h>
void swap(int *v1, int *v2)
{
    int tmp;
    tmp = *v1;
    *v1 = *v2;
    *v2 = tmp;
}
int main(void)
{
    int n1 = 3, n2 = 418;
    swap(&n1, &n2);
    printf("n1 = %4.2f, n2 = %4.2f\n", n1, n2);
    return 0;
}
```



Ett till exempel

code/swap.c

```c
#include <stdio.h>
void swap(int *v1, int *v2)
{
    int tmp;
    tmp = *v1;
    *v1 = *v2;
    *v2 = tmp;
}
int main(void)
{
    int n1 = 3, n2 = 418;
    swap(&n1, &n2);
    printf("n1 = %4.2f, n2 = %4.2f\n", n1, n2);
    return 0;
}
```

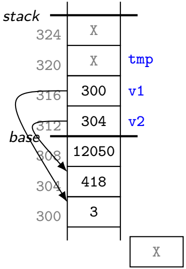


[redacted]

Ett till exempel

code/swap.c

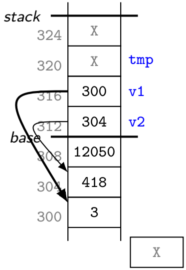
```c
#include <stdio.h>
void swap(int *v1, int *v2)
{
    int tmp;
    tmp = *v1;
    *v1 = *v2;
    *v2 = tmp;
}
int main(void)
{
    int n1 = 3, n2 = 418;
    swap(&n1, &n2);
    printf("n1 = %4.2f, n2 = %4.2f\n", n1, n2);
    return 0;
}
```



Ett till exempel

code/swap.c

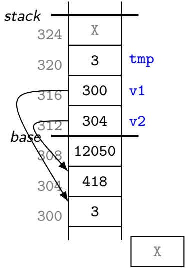
```c
#include <stdio.h>
void swap(int *v1, int *v2)
{
    int tmp;
    tmp = *v1;
    *v1 = *v2;
    *v2 = tmp;
}
int main(void)
{
    int n1 = 3, n2 = 418;
    swap(&n1, &n2);
    printf("n1 = %4.2f, n2 = %4.2f\n", n1, n2);
    return 0;
}
```



Ett till exempel

code/swap.c

```c
#include <stdio.h>
void swap(int *v1, int *v2)
{
    int tmp;
    tmp = *v1;
    *v1 = *v2;
    *v2 = tmp;
}
int main(void)
{
    int n1 = 3, n2 = 418;
    swap(&n1, &n2);
    printf("n1 = %4.2f, n2 = %4.2f\n", n1, n2);
    return 0;
}
```

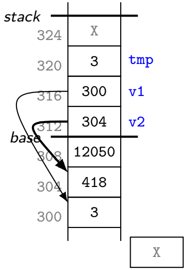


[redacted]

Ett till exempel

code/swap.c

```c
#include <stdio.h>
void swap(int *v1, int *v2)
{
    int tmp;
    tmp = *v1;
    *v1 = *v2;
    *v2 = tmp;
}
int main(void)
{
    int n1 = 3, n2 = 418;
    swap(&n1, &n2);
    printf("n1 = %4.2f, n2 = %4.2f\n", n1, n2);
    return 0;
}
```

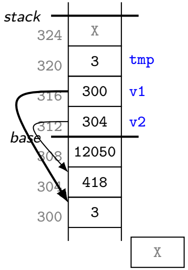


[redacted]

Ett till exempel

code/swap.c

```
#include <stdio.h>
void swap(int *v1, int *v2)
{
    int tmp;
    tmp = *v1;
    *v1 = *v2;
    *v2 = tmp;
}
int main(void)
{
    int n1 = 3, n2 = 418;
    swap(&n1, &n2);
    printf("n1 = %4.2f, n2 = %4.2f\n", n1, n2);
    return 0;
}
```

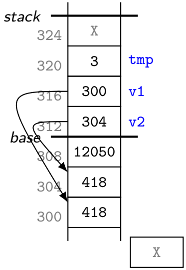


[redacted]

Ett till exempel

code/swap.c

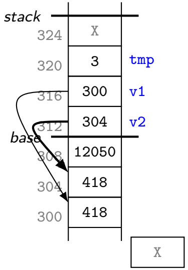
```c
#include <stdio.h>
void swap(int *v1, int *v2)
{
    int tmp;
    tmp = *v1;
    *v1 = *v2;
    *v2 = tmp;
}
int main(void)
{
    int n1 = 3, n2 = 418;
    swap(&n1, &n2);
    printf("n1 = %4.2f, n2 = %4.2f\n", n1, n2);
    return 0;
}
```



Ett till exempel

code/swap.c

```c
#include <stdio.h>
void swap(int *v1, int *v2)
{
    int tmp;
    tmp = *v1;
    *v1 = *v2;
    *v2 = tmp;
}
int main(void)
{
    int n1 = 3, n2 = 418;
    swap(&n1, &n2);
    printf("n1 = %4.2f, n2 = %4.2f\n", n1, n2);
    return 0;
}
```

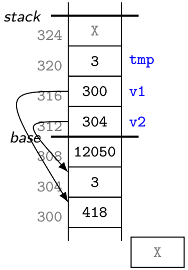


[redacted]

Ett till exempel

code/swap.c

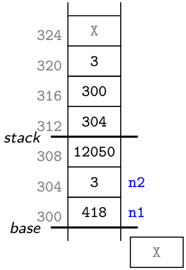
```c
#include <stdio.h>
void swap(int *v1, int *v2)
{
    int tmp;
    tmp = *v1;
    *v1 = *v2;
    *v2 = tmp;
}
int main(void)
{
    int n1 = 3, n2 = 418;
    swap(&n1, &n2);
    printf("n1 = %4.2f, n2 = %4.2f\n", n1, n2);
    return 0;
}
```



Ett till exempel

code/swap.c

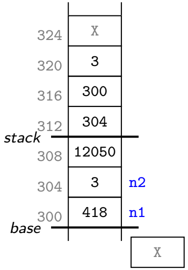
```c
#include <stdio.h>
void swap(int *v1, int *v2)
{
    int tmp;
    tmp = *v1;
    *v1 = *v2;
    *v2 = tmp;
}
int main(void)
{
    int n1 = 3, n2 = 418;
    swap(&n1, &n2);
    printf("n1 = %4.2f, n2 = %4.2f\n", n1, n2);
    return 0;
}
```



Ett till exempel

code/swap.c

```c
#include <stdio.h>
void swap(int *v1, int *v2)
{
    int tmp;
    tmp = *v1;
    *v1 = *v2;
    *v2 = tmp;
}
int main(void)
{
    int n1 = 3, n2 = 418;
    swap(&n1, &n2);
    printf("n1 = %4.2f, n2 = %4.2f\n", n1, n2);
    return 0;
}
```

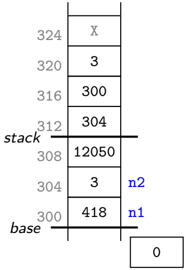



```
n1 = 418, n2 = 3
```

Ett till exempel

code/swap.c

```c
#include <stdio.h>
void swap(int *v1, int *v2)
{
    int tmp;
    tmp = *v1;
    *v1 = *v2;
    *v2 = tmp;
}
int main(void)
{
    int n1 = 3, n2 = 418;
    swap(&n1, &n2);
    printf("n1 = %4.2f, n2 = %4.2f\n", n1, n2);
    return 0;
}
```

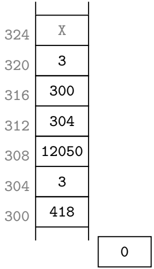



```
n1 = 418, n2 = 3
```

Ett till exempel

code/swap.c

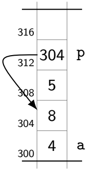
```c
#include <stdio.h>
void swap(int *v1, int *v2)
{
    int tmp;
    tmp = *v1;
    *v1 = *v2;
    *v2 = tmp;
}
int main(void)
{
    int n1 = 3, n2 = 418;
    swap(&n1, &n2);
    printf("n1 = %4.2f, n2 = %4.2f\n", n1, n2);
    return 0;
}
```




```
n1 = 418, n2 = 3
```

Pekare och fält

- I C så är ett fält och en pekare nästan samma sak
  - En fältvariabel kan inte pekas om
- `int a[3] = {4, 8, 5};`
- `int *p = a + 1;`
- Både pekare och arrayer kan indexeras
  - `p[1]` och `*(p+1)` är syntaktiskt ekvivalenta
    - och refererar till samma minne som `a[2]`
  - `p[0]` är syntaktiskt ekvivalent med `*p`
- C har inget stöd för kontroll av fältgränser
  - `a[3] = 800` följt av `*p = 10` skulle kunna bli intressant...



Pekare och poster (1)

- Vi kan också ha pekare till poster (struct)

code/struct_on_stack.c
```
typedef struct node {
    int val;
    struct node *next;
} node;
```

- Ovanstående kod definierar typen `struct node` med fälten:
  - `val` av typen `int`
  - `next` av typen `struct node *`
    - `next` är alltså en pekare till en variabel av samma typ
- Dessutom deklarerar `typedef`-satsen att typen `node` är ett annat namn på `struct node`
- Givet definitionen ovan så kan vi deklarera variabler av typen `node` och `node *`:

```
node n1;
node *n = &n1;
```

Pekare och poster (2)

- code/struct_on_stack.c

```c
typedef struct node {
    int val;
    struct node *next;
} node;
```

- Referenser till fälten i en `struct` görs med punkt-operatorn
  - `n1.val = 22;`
  - `n1.next = NULL;`
- Referenser till fälten i en `struct` via en pekare till `struct` kan också göras med operatorn `->` (”minus, större än”)
  - Följande två uttryck är ekvivalenta
    - `(*n).val = 22;`
    - `n->val = 22;`
  - Den sista formen är vanligast

Länkade strukturer

Länkade listor

1. Exempel med noder som lokala variabler i main (`struct-on-stack`)
2. Exempel med noder med typat värde som skapas dynamiskt (`dynamic-struct`)
3. Exempel med noder med generiskt pekarvärde som använder kodbasen
   - 3.1 Lista med `kill_function` (`list-with-killhandler`)
   - 3.2 Lista utan `kill_function` (`list-no-killhandler`)

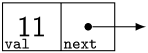
Lista 1: Typad payload, noder som lokala variabler (1)

- I det första exemplet har vi en post-typ som lagrar ”payload” inuti posten i form av ett heltal

code/struct-on-stack.c

```
typedef struct node {
    int val; // typed payload of fixed size inside struct
    struct node *next;
} node;
```

- Denna struct motsvarar en 1-cell där värdet är en `int`:



Lista 1: Typad payload, noder som lokala variabler (2)

- Då typen på `val` är känd kan vi t.ex. anropa `printf` direkt med värdet

code/struct-on-stack.c

```c
typedef struct node {
    int val; // typed payload of fixed size inside struct
    struct node *next;
} node;
```

code/struct-on-stack.c

```c
void print_list(const node *p)
{
    printf("( ");
    while (p != NULL) {
        printf("%d ", p->val); // we know the type of val
        p = p->next;
    }
    printf(")\n");
}
```

Lista 1: Typad payload, noder som lokala variabler (3)

- I exemplet så lagrar vi elementen som ska läggas i listan som lokala variabler, dvs. på stacken

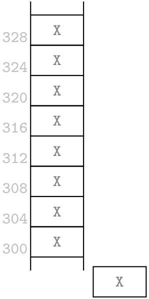
code/struct-on-stack.c

```c
int main(void)
{
    node n1; // automatic allocation on stack
    node n2;
    node *head;
    n2.val = 21;
    n2.next = NULL;
    n1.val = 20;
    n1.next = &n2;
    head = &n1;
    print_list(head);
    return 0;
} // automatic cleanup
```

- Allokering och avallokering kommer att ske automatiskt

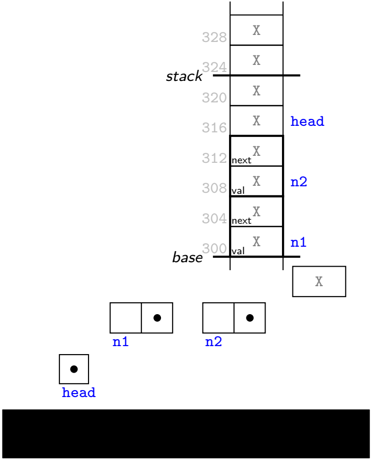
Lista 1: Visualisering

```c
#include <stdio.h>
typedef struct node {
    int val; // typed payload of fixed size inside struct
    struct node *next;
} node;
void print_list(const node *p)
{
    printf("( ");
    while (p != NULL) {
        printf("%d ", p->val); // we know the type of val
        p = p->next;
    }
    printf(")\n");
}
int main(void)
{
    node n1; // automatic allocation on stack
    node n2;
    node *head;
    n2.val = 21;
    n2.next = NULL;
    n1.val = 20;
    n1.next = &n2;
    head = &n1;
    print_list(head);
    return 0;
} // automatic cleanup
```



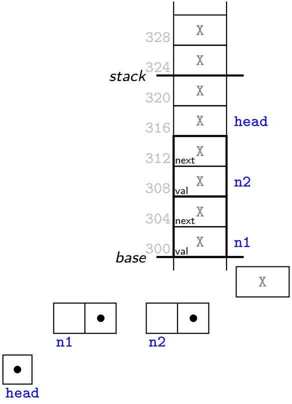
Lista 1: Visualisering

```c
#include <stdio.h>
typedef struct node {
    int val; // typed payload of fixed size inside struct
    struct node *next;
} node;
void print_list(const node *p)
{
    printf("( ");
    while (p != NULL) {
        printf("%d ", p->val); // we know the type of val
        p = p->next;
    }
    printf(")\n");
}
int main(void)
{
    node n1; // automatic allocation on stack
    node n2;
    node *head;
    n2.val = 21;
    n2.next = NULL;
    n1.val = 20;
    n1.next = &n2;
    head = &n1;
    print_list(head);
    return 0;
} // automatic cleanup
```



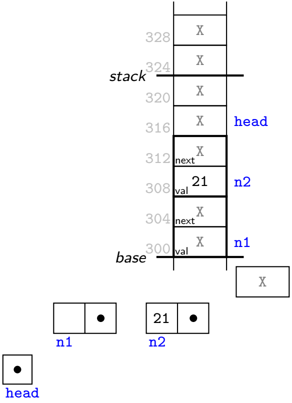
Lista 1: Visualisering

```c
#include <stdio.h>
typedef struct node {
    int val; // typed payload of fixed size inside struct
    struct node *next;
} node;
void print_list(const node *p)
{
    printf("( ");
    while (p != NULL) {
        printf("%d ", p->val); // we know the type of val
        p = p->next;
    }
    printf(")\n");
}
int main(void)
{
    node n1; // automatic allocation on stack
    node n2;
    node *head;
    n2.val = 21;
    n2.next = NULL;
    n1.val = 20;
    n1.next = &n2;
    head = &n1;
    print_list(head);
    return 0;
} // automatic cleanup
```



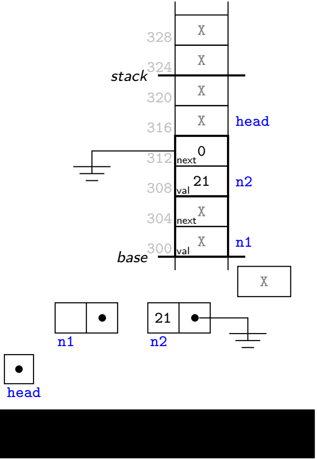
Lista 1: Visualisering

```c
#include <stdio.h>
typedef struct node {
    int val; // typed payload of fixed size inside struct
    struct node *next;
} node;
void print_list(const node *p)
{
    printf("( ");
    while (p != NULL) {
        printf("%d ", p->val); // we know the type of val
        p = p->next;
    }
    printf(")\n");
}
int main(void)
{
    node n1; // automatic allocation on stack
    node n2;
    node *head;
    n2.val = 21;
    n2.next = NULL;
    n1.val = 20;
    n1.next = &n2;
    head = &n1;
    print_list(head);
    return 0;
} // automatic cleanup
```



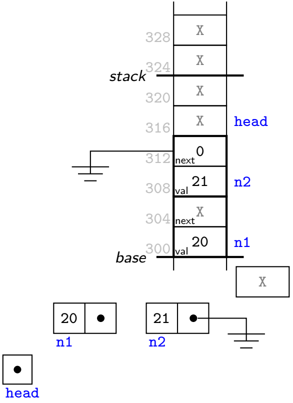
Lista 1: Visualisering

```c
#include <stdio.h>
typedef struct node {
    int val; // typed payload of fixed size inside struct
    struct node *next;
} node;
void print_list(const node *p)
{
    printf("( ");
    while (p != NULL) {
        printf("%d ", p->val); // we know the type of val
        p = p->next;
    }
    printf(")\n");
}
int main(void)
{
    node n1; // automatic allocation on stack
    node n2;
    node *head;
    n2.val = 21;
    n2.next = NULL;
    n1.val = 20;
    n1.next = &n2;
    head = &n1;
    print_list(head);
    return 0;
} // automatic cleanup
```



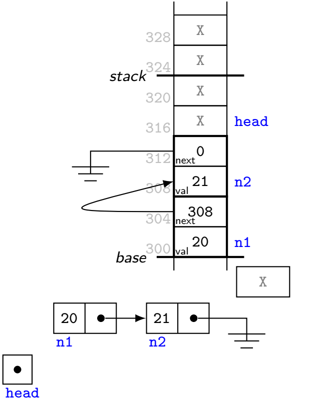
Lista 1: Visualisering

```c
#include <stdio.h>
typedef struct node {
    int val; // typed payload of fixed size inside struct
    struct node *next;
} node;
void print_list(const node *p)
{
    printf("( ");
    while (p != NULL) {
        printf("%d ", p->val); // we know the type of val
        p = p->next;
    }
    printf(")\n");
}
int main(void)
{
    node n1; // automatic allocation on stack
    node n2;
    node *head;
    n2.val = 21;
    n2.next = NULL;
    n1.val = 20;
    n1.next = &n2;
    head = &n1;
    print_list(head);
    return 0;
} // automatic cleanup
```



Lista 1: Visualisering

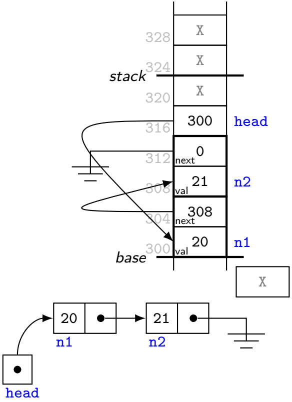
```c
#include <stdio.h>
typedef struct node {
    int val; // typed payload of fixed size inside struct
    struct node *next;
} node;
void print_list(const node *p)
{
    printf("( ");
    while (p != NULL) {
        printf("%d ", p->val); // we know the type of val
        p = p->next;
    }
    printf(")\n");
}
int main(void)
{
    node n1; // automatic allocation on stack
    node n2;
    node *head;
    n2.val = 21;
    n2.next = NULL;
    n1.val = 20;
    n1.next = &n2;
    head = &n1;
    print_list(head);
    return 0;
} // automatic cleanup
```



[redacted]

Lista 1: Visualisering

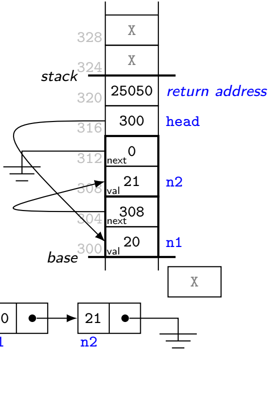
```c
#include <stdio.h>
typedef struct node {
    int val; // typed payload of fixed size inside struct
    struct node *next;
} node;
void print_list(const node *p)
{
    printf("( ");
    while (p != NULL) {
        printf("%d ", p->val); // we know the type of val
        p = p->next;
    }
    printf(")\n");
}
int main(void)
{
    node n1; // automatic allocation on stack
    node n2;
    node *head;
    n2.val = 21;
    n2.next = NULL;
    n1.val = 20;
    n1.next = &n2;
    head = &n1;
    print_list(head);
    return 0;
} // automatic cleanup
```



[redacted]

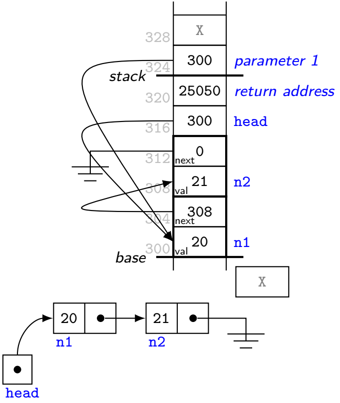
Lista 1: Visualisering

```c
#include <stdio.h>
typedef struct node {
    int val; // typed payload of fixed size inside struct
    struct node *next;
} node;
void print_list(const node *p)
{
    printf("( ");
    while (p != NULL) {
        printf("%d ", p->val); // we know the type of val
        p = p->next;
    }
    printf(")\n");
}
int main(void)
{
    node n1; // automatic allocation on stack
    node n2;
    node *head;
    n2.val = 21;
    n2.next = NULL;
    n1.val = 20;
    n1.next = &n2;
    head = &n1;
    print_list(head);
    return 0;
} // automatic cleanup
```



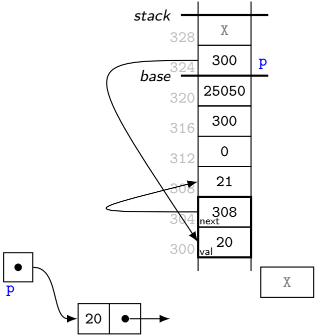
Lista 1: Visualisering

```c
#include <stdio.h>
typedef struct node {
    int val; // typed payload of fixed size inside struct
    struct node *next;
} node;
void print_list(const node *p)
{
    printf("( ");
    while (p != NULL) {
        printf("%d ", p->val); // we know the type of val
        p = p->next;
    }
    printf(")\n");
}
int main(void)
{
    node n1; // automatic allocation on stack
    node n2;
    node *head;
    n2.val = 21;
    n2.next = NULL;
    n1.val = 20;
    n1.next = &n2;
    head = &n1;
    print_list(head);
    return 0;
} // automatic cleanup
```



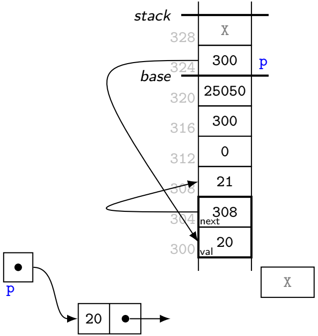
Lista 1: Visualisering

```c
#include <stdio.h>
typedef struct node {
    int val; // typed payload of fixed size inside struct
    struct node *next;
} node;
void print_list(const node *p)
{
    printf("( ");
    while (p != NULL) {
        printf("%d ", p->val); // we know the type of val
        p = p->next;
    }
    printf(")\n");
}
int main(void)
{
    node n1; // automatic allocation on stack
    node n2;
    node *head;
    n2.val = 21;
    n2.next = NULL;
    n1.val = 20;
    n1.next = &n2;
    head = &n1;
    print_list(head);
    return 0;
} // automatic cleanup
```



Lista 1: Visualisering

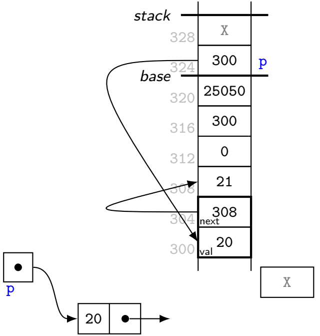
```c
#include <stdio.h>
typedef struct node {
    int val; // typed payload of fixed size inside struct
    struct node *next;
} node;
void print_list(const node *p)
{
    printf("( ");
    while (p != NULL) {
        printf("%d ", p->val); // we know the type of val
        p = p->next;
    }
    printf(")\n");
}
int main(void)
{
    node n1; // automatic allocation on stack
    node n2;
    node *head;
    n2.val = 21;
    n2.next = NULL;
    n1.val = 20;
    n1.next = &n2;
    head = &n1;
    print_list(head);
    return 0;
} // automatic cleanup
```



[redacted]

Lista 1: Visualisering

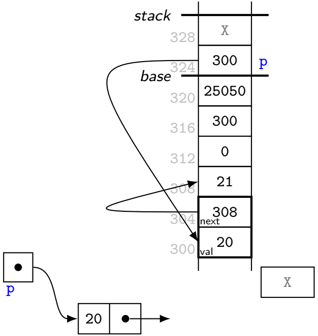
```c
#include <stdio.h>
typedef struct node {
    int val; // typed payload of fixed size inside struct
    struct node *next;
} node;
void print_list(const node *p)
{
    printf("( ");
    while (p != NULL) {
        printf("%d ", p->val); // we know the type of val
        p = p->next;
    }
    printf(")\n");
}
int main(void)
{
    node n1; // automatic allocation on stack
    node n2;
    node *head;
    n2.val = 21;
    n2.next = NULL;
    n1.val = 20;
    n1.next = &n2;
    head = &n1;
    print_list(head);
    return 0;
} // automatic cleanup
```




```
(
```

Lista 1: Visualisering

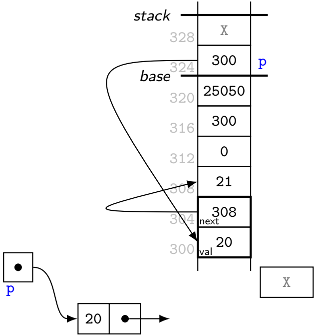
```c
#include <stdio.h>
typedef struct node {
    int val; // typed payload of fixed size inside struct
    struct node *next;
} node;
void print_list(const node *p)
{
    printf("( ");
    while (p != NULL) {
        printf("%d ", p->val); // we know the type of val
        p = p->next;
    }
    printf(")\n");
}
int main(void)
{
    node n1; // automatic allocation on stack
    node n2;
    node *head;
    n2.val = 21;
    n2.next = NULL;
    n1.val = 20;
    n1.next = &n2;
    head = &n1;
    print_list(head);
    return 0;
} // automatic cleanup
```




```
(
```

Lista 1: Visualisering

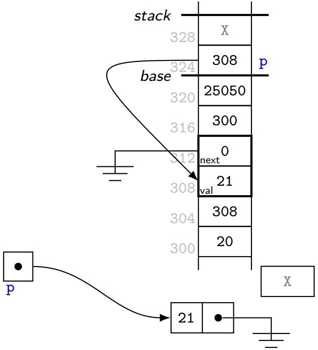
```c
#include <stdio.h>
typedef struct node {
    int val; // typed payload of fixed size inside struct
    struct node *next;
} node;
void print_list(const node *p)
{
    printf("( ");
    while (p != NULL) {
        printf("%d ", p->val); // we know the type of val
        p = p->next;
    }
    printf(")\n");
}
int main(void)
{
    node n1; // automatic allocation on stack
    node n2;
    node *head;
    n2.val = 21;
    n2.next = NULL;
    n1.val = 20;
    n1.next = &n2;
    head = &n1;
    print_list(head);
    return 0;
} // automatic cleanup
```




```
( 20
```

Lista 1: Visualisering

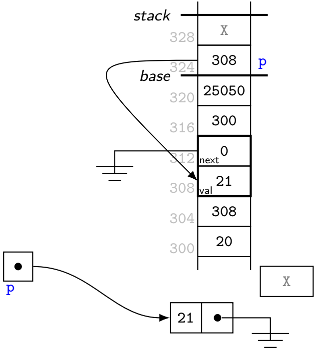
```c
#include <stdio.h>
typedef struct node {
    int val; // typed payload of fixed size inside struct
    struct node *next;
} node;
void print_list(const node *p)
{
    printf("( ");
    while (p != NULL) {
        printf("%d ", p->val); // we know the type of val
        p = p->next;
    }
    printf(")\n");
}
int main(void)
{
    node n1; // automatic allocation on stack
    node n2;
    node *head;
    n2.val = 21;
    n2.next = NULL;
    n1.val = 20;
    n1.next = &n2;
    head = &n1;
    print_list(head);
    return 0;
} // automatic cleanup
```



(20

Lista 1: Visualisering

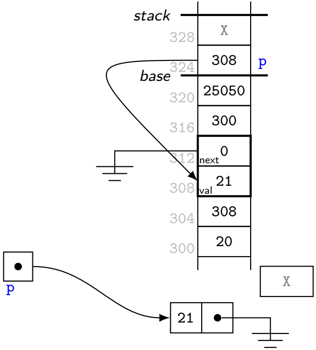
```c
#include <stdio.h>
typedef struct node {
    int val; // typed payload of fixed size inside struct
    struct node *next;
} node;
void print_list(const node *p)
{
    printf("( ");
    while (p != NULL) {
        printf("%d ", p->val); // we know the type of val
        p = p->next;
    }
    printf(")\n");
}
int main(void)
{
    node n1; // automatic allocation on stack
    node n2;
    node *head;
    n2.val = 21;
    n2.next = NULL;
    n1.val = 20;
    n1.next = &n2;
    head = &n1;
    print_list(head);
    return 0;
} // automatic cleanup
```




```
( 20
```

Lista 1: Visualisering

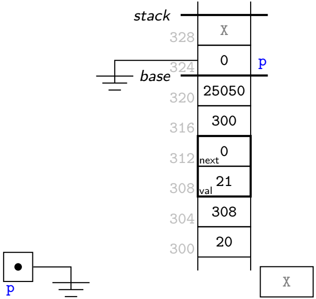
```c
#include <stdio.h>
typedef struct node {
    int val; // typed payload of fixed size inside struct
    struct node *next;
} node;
void print_list(const node *p)
{
    printf("( ");
    while (p != NULL) {
        printf("%d ", p->val); // we know the type of val
        p = p->next;
    }
    printf(")\n");
}
int main(void)
{
    node n1; // automatic allocation on stack
    node n2;
    node *head;
    n2.val = 21;
    n2.next = NULL;
    n1.val = 20;
    n1.next = &n2;
    head = &n1;
    print_list(head);
    return 0;
} // automatic cleanup
```




```
( 20 21
```

Lista 1: Visualisering

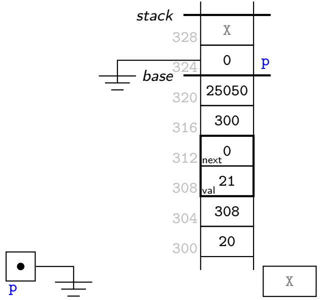
```c
#include <stdio.h>
typedef struct node {
    int val; // typed payload of fixed size inside struct
    struct node *next;
} node;
void print_list(const node *p)
{
    printf("( ");
    while (p != NULL) {
        printf("%d ", p->val); // we know the type of val
        p = p->next;
    }
    printf(")\n");
}
int main(void)
{
    node n1; // automatic allocation on stack
    node n2;
    node *head;
    n2.val = 21;
    n2.next = NULL;
    n1.val = 20;
    n1.next = &n2;
    head = &n1;
    print_list(head);
    return 0;
} // automatic cleanup
```




```
( 20 21
```

Lista 1: Visualisering

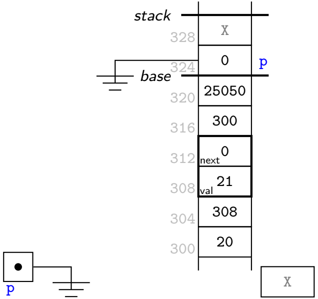
```c
#include <stdio.h>
typedef struct node {
    int val; // typed payload of fixed size inside struct
    struct node *next;
} node;
void print_list(const node *p)
{
    printf("( ");
    while (p != NULL) {
        printf("%d ", p->val); // we know the type of val
        p = p->next;
    }
    printf(")\n");
}
int main(void)
{
    node n1; // automatic allocation on stack
    node n2;
    node *head;
    n2.val = 21;
    n2.next = NULL;
    n1.val = 20;
    n1.next = &n2;
    head = &n1;
    print_list(head);
    return 0;
} // automatic cleanup
```




```
( 20 21
```

Lista 1: Visualisering

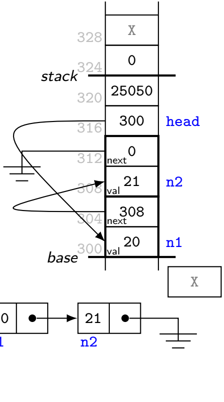
```c
#include <stdio.h>
typedef struct node {
    int val; // typed payload of fixed size inside struct
    struct node *next;
} node;
void print_list(const node *p)
{
    printf("( ");
    while (p != NULL) {
        printf("%d ", p->val); // we know the type of val
        p = p->next;
    }
    printf(")\n");
}
int main(void)
{
    node n1; // automatic allocation on stack
    node n2;
    node *head;
    n2.val = 21;
    n2.next = NULL;
    n1.val = 20;
    n1.next = &n2;
    head = &n1;
    print_list(head);
    return 0;
} // automatic cleanup
```



(20 21)

Lista 1: Visualisering

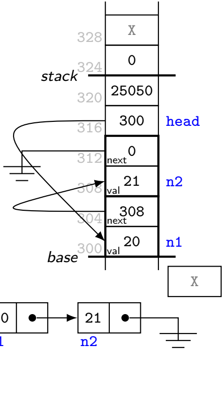
```c
#include <stdio.h>
typedef struct node {
    int val; // typed payload of fixed size inside struct
    struct node *next;
} node;
void print_list(const node *p)
{
    printf("( ");
    while (p != NULL) {
        printf("%d ", p->val); // we know the type of val
        p = p->next;
    }
    printf(")\n");
}
int main(void)
{
    node n1; // automatic allocation on stack
    node n2;
    node *head;
    n2.val = 21;
    n2.next = NULL;
    n1.val = 20;
    n1.next = &n2;
    head = &n1;
    print_list(head);
    return 0;
} // automatic cleanup
```



(20 21)

Lista 1: Visualisering

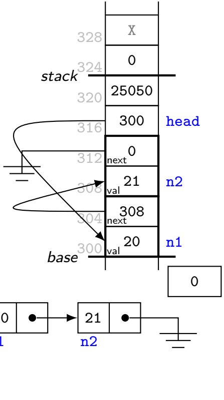
```c
#include <stdio.h>
typedef struct node {
    int val; // typed payload of fixed size inside struct
    struct node *next;
} node;
void print_list(const node *p)
{
    printf("( ");
    while (p != NULL) {
        printf("%d ", p->val); // we know the type of val
        p = p->next;
    }
    printf(")\n");
}
int main(void)
{
    node n1; // automatic allocation on stack
    node n2;
    node *head;
    n2.val = 21;
    n2.next = NULL;
    n1.val = 20;
    n1.next = &n2;
    head = &n1;
    print_list(head);
    return 0;
} // automatic cleanup
```




```
( 20 21 )
```

Lista 1: Visualisering

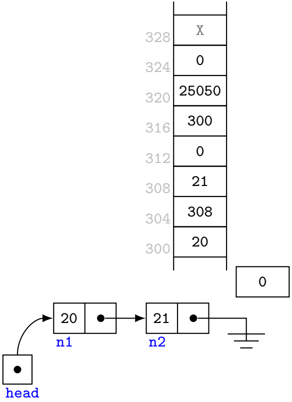
```c
#include <stdio.h>
typedef struct node {
    int val; // typed payload of fixed size inside struct
    struct node *next;
} node;
void print_list(const node *p)
{
    printf("( ");
    while (p != NULL) {
        printf("%d ", p->val); // we know the type of val
        p = p->next;
    }
    printf(")\n");
}
int main(void)
{
    node n1; // automatic allocation on stack
    node n2;
    node *head;
    n2.val = 21;
    n2.next = NULL;
    n1.val = 20;
    n1.next = &n2;
    head = &n1;
    print_list(head);
    return 0;
} // automatic cleanup
```




```
( 20 21 )
```

Lista 1: Visualisering

```c
#include <stdio.h>
typedef struct node {
    int val; // typed payload of fixed size inside struct
    struct node *next;
} node;
void print_list(const node *p)
{
    printf("( ");
    while (p != NULL) {
        printf("%d ", p->val); // we know the type of val
        p = p->next;
    }
    printf(")\n");
}
int main(void)
{
    node n1; // automatic allocation on stack
    node n2;
    node *head;
    n2.val = 21;
    n2.next = NULL;
    n1.val = 20;
    n1.next = &n2;
    head = &n1;
    print_list(head);
    return 0;
} // automatic cleanup
```



(20 21)

Begränsningar

- Att ha noderna på stacken innebär flera begränsningar
  1. Vi har inte kontroll över minneshanteringen
     - Minne reserveras automatiskt på stacken när en funktion startar
     - Minne återlämnas automatiskt när en funktion slutar
  2. Antalet variabler är statiskt — vi kan inte allokera godtyckligt många
- För detta behöver vi dynamisk minneshantering
  - Vi reserverar (allokerar) minne när det behövs
  - Vi lämnar tillbaka (deallokerar, frigör) minne när vi inte längre behöver det

Dynamisk minneshantering i C

Dynamisk minneshantering i C (1)

- För att reservera minne i C används de inbyggda funktionerna `malloc` och `calloc`:

stdlib.h
```
void *malloc(size_t size);
void *calloc(size_t nelem, size_t size);
```

  - Bägge returnerar en pekare till reserverat minne om OK, annars `NULL`
  - `malloc` reserverar `size` bytes med minne
  - `calloc` reserverar `nelem` element av storleken `size` bytes och initierar det reserverade minnet till 0
  - Minnet reserveras på heapen
- Typen **`size_t`** är en heltalstyp stor nog att rymma den största storleken på ett objekt i ditt system

Dynamisk minneshantering i C (2)

- Reservera minne till en heltalsvektor med 10 element:

```
int *v1 = malloc(10 * sizeof(int));
int *v2 = calloc(10, sizeof(int));
int *v3 = malloc(10 * sizeof(*v3));
int *v4 = calloc(10, sizeof(*v4));
```

  - Operatorn `sizeof` returnerar storleken på argumentet i bytes

Dynamisk minneshantering i C (3)

- För att återlämna minne i C finns funktionen `free`

stdlib.h
```
void free(void *p);
```

  - Notera att `free` behöver bara pekaren, inte storleken på det reserverade minnesblocket
- Varje pekare som returneras från `malloc` eller `calloc` måste skickas tillbaka till `free`
  - Annars får vi en s.k. minnesläcka: Minnet är markerat som reserverat och kommer aldrig att lämnas tillbaka

Minnesballonger

- En hjälpsam analogi är att tänka på minnesblocken som helium-ballonger
- Pekarna är snören till ballongerna
  - Vi kan ha flera snören till varje ballong
  - Ett snöre kan vara fastknutet i en lokal variabel
  - Vi kan knyta fast ett snöre i en annan variabel, t.ex. en nod
    - Noden måste vara en lokal variabel eller vara fastknuten
- Ballongen måste vara fastknuten i minst en lokal variabel
  - Tappar vi alla snören tappar vi ballongen och då flyger den iväg (minnesläcka)
- Ballongen måste återlämnas innan `main` avslutas

Lista 2: Typad payload, dynamiska noder

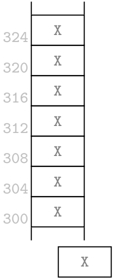
- I lista 2 allokeras och avallokeras noderna dynamiskt

code/dynamic-struct.c

```c
int main(void)
{
    node *n, *head = NULL;
    for (int i = 0; i < 2; i++) {
        n = malloc(sizeof(*n));
        n->val = 21 - i;
        n->next = head;
        head = n;
    }
    print_list(head);
    n = head; // Explicit cleanup
    while (n != NULL) {
        node *p = n->next;
        free(n);
        n = p;
    }
    return 0;
}
```

Lista 2: Typad payload, dynamiska noder

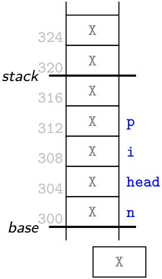
```c
typedef struct node {
    int val; // typed payload of fixed size inside struct
    struct node *next;
} node;
void print_list(const node *p)
{
    printf("( ");
    while (p != NULL) {
        printf("%d ", p->val); // we know the type of val
        p = p->next;
    }
    printf(")\n");
}
int main(void)
{
    node *n, *head = NULL;
    for (int i = 0; i < 2; i++) {
        n = malloc(sizeof(*n));
        n->val = 21 - i;
        n->next = head;
        head = n;
    }
    print_list(head);
    n = head; // Explicit cleanup
    while (n != NULL) {
        node *p = n->next;
        free(n);
        n = p;
    }
    return 0;
}
```



[redacted]

Lista 2: Typad payload, dynamiska noder

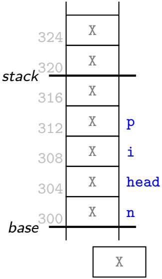
```c
typedef struct node {
    int val; // typed payload of fixed size inside struct
    struct node *next;
} node;
void print_list(const node *p)
{
    printf("( ");
    while (p != NULL) {
        printf("%d ", p->val); // we know the type of val
        p = p->next;
    }
    printf(")\n");
}
int main(void)
{
    node *n, *head = NULL;
    for (int i = 0; i < 2; i++) {
        n = malloc(sizeof(*n));
        n->val = 21 - i;
        n->next = head;
        head = n;
    }
    print_list(head);
    n = head; // Explicit cleanup
    while (n != NULL) {
        node *p = n->next;
        free(n);
        n = p;
    }
    return 0;
}
```



[redacted]

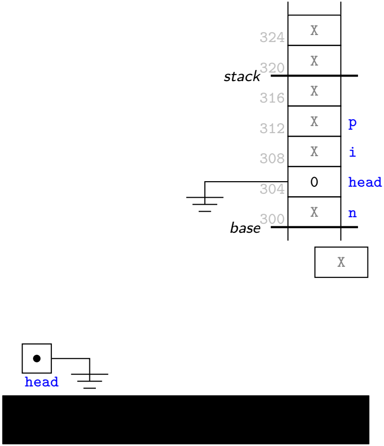
Lista 2: Typad payload, dynamiska noder

```c
typedef struct node {
    int val; // typed payload of fixed size inside struct
    struct node *next;
} node;
void print_list(const node *p)
{
    printf("( ");
    while (p != NULL) {
        printf("%d ", p->val); // we know the type of val
        p = p->next;
    }
    printf(")\n");
}
int main(void)
{
    node *n, *head = NULL;
    for (int i = 0; i < 2; i++) {
        n = malloc(sizeof(*n));
        n->val = 21 - i;
        n->next = head;
        head = n;
    }
    print_list(head);
    n = head; // Explicit cleanup
    while (n != NULL) {
        node *p = n->next;
        free(n);
        n = p;
    }
    return 0;
}
```



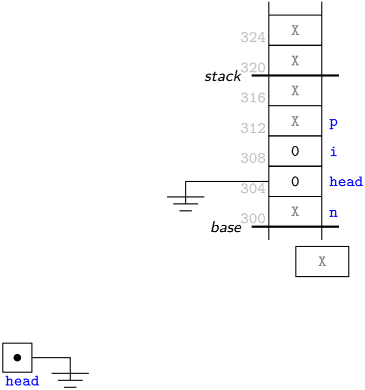
Lista 2: Typad payload, dynamiska noder

```c
typedef struct node {
    int val; // typed payload of fixed size inside struct
    struct node *next;
} node;
void print_list(const node *p)
{
    printf("( ");
    while (p != NULL) {
        printf("%d ", p->val); // we know the type of val
        p = p->next;
    }
    printf(")\n");
}
int main(void)
{
    node *n, *head = NULL;
    for (int i = 0; i < 2; i++) {
        n = malloc(sizeof(*n));
        n->val = 21 - i;
        n->next = head;
        head = n;
    }
    print_list(head);
    n = head; // Explicit cleanup
    while (n != NULL) {
        node *p = n->next;
        free(n);
        n = p;
    }
    return 0;
}
```



Lista 2: Typad payload, dynamiska noder

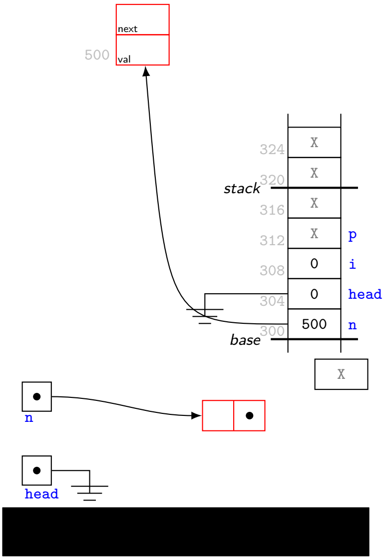
```c
typedef struct node {
    int val; // typed payload of fixed size inside struct
    struct node *next;
} node;
void print_list(const node *p)
{
    printf("( ");
    while (p != NULL) {
        printf("%d ", p->val); // we know the type of val
        p = p->next;
    }
    printf(")\n");
}
int main(void)
{
    node *n, *head = NULL;
    for (int i = 0; i < 2; i++) {
        n = malloc(sizeof(*n));
        n->val = 21 - i;
        n->next = head;
        head = n;
    }
    print_list(head);
    n = head; // Explicit cleanup
    while (n != NULL) {
        node *p = n->next;
        free(n);
        n = p;
    }
    return 0;
}
```



[redacted]

Lista 2: Typad payload, dynamiska noder

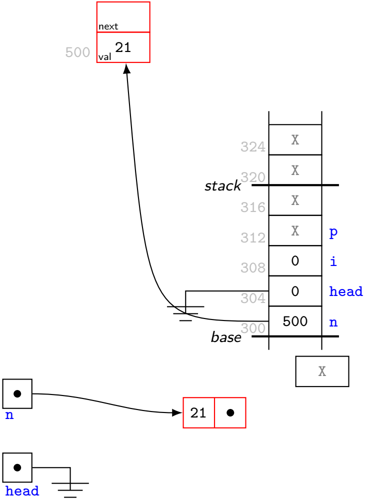
```c
typedef struct node {
    int val; // typed payload of fixed size inside struct
    struct node *next;
} node;
void print_list(const node *p)
{
    printf("( ");
    while (p != NULL) {
        printf("%d ", p->val); // we know the type of val
        p = p->next;
    }
    printf(")\n");
}
int main(void)
{
    node *n, *head = NULL;
    for (int i = 0; i < 2; i++) {
        n = malloc(sizeof(*n));
        n->val = 21 - i;
        n->next = head;
        head = n;
    }
    print_list(head);
    n = head; // Explicit cleanup
    while (n != NULL) {
        node *p = n->next;
        free(n);
        n = p;
    }
    return 0;
}
```



[redacted]

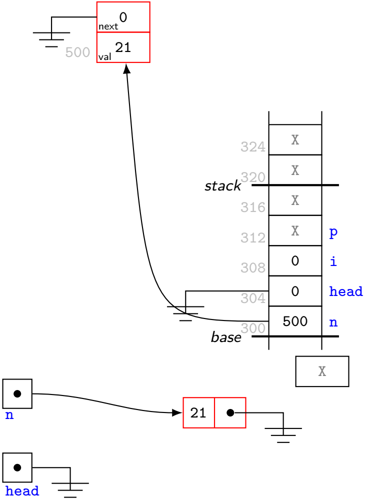
Lista 2: Typad payload, dynamiska noder

```c
typedef struct node {
    int val; // typed payload of fixed size inside struct
    struct node *next;
} node;
void print_list(const node *p)
{
    printf("( ");
    while (p != NULL) {
        printf("%d ", p->val); // we know the type of val
        p = p->next;
    }
    printf(")\n");
}
int main(void)
{
    node *n, *head = NULL;
    for (int i = 0; i < 2; i++) {
        n = malloc(sizeof(*n));
        n->val = 21 - i;
        n->next = head;
        head = n;
    }
    print_list(head);
    n = head; // Explicit cleanup
    while (n != NULL) {
        node *p = n->next;
        free(n);
        n = p;
    }
    return 0;
}
```



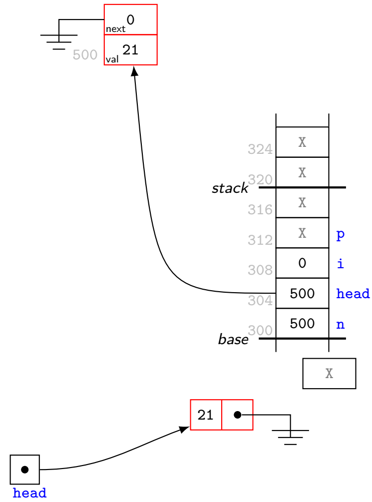
Lista 2: Typad payload, dynamiska noder

```c
typedef struct node {
    int val; // typed payload of fixed size inside struct
    struct node *next;
} node;
void print_list(const node *p)
{
    printf("( ");
    while (p != NULL) {
        printf("%d ", p->val); // we know the type of val
        p = p->next;
    }
    printf(")\n");
}
int main(void)
{
    node *n, *head = NULL;
    for (int i = 0; i < 2; i++) {
        n = malloc(sizeof(*n));
        n->val = 21 - i;
        n->next = head;
        head = n;
    }
    print_list(head);
    n = head; // Explicit cleanup
    while (n != NULL) {
        node *p = n->next;
        free(n);
        n = p;
    }
    return 0;
}
```



Lista 2: Typad payload, dynamiska noder

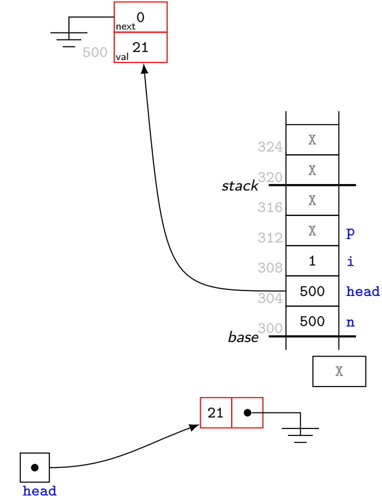
```c
typedef struct node {
    int val; // typed payload of fixed size inside struct
    struct node *next;
} node;
void print_list(const node *p)
{
    printf("( ");
    while (p != NULL) {
        printf("%d ", p->val); // we know the type of val
        p = p->next;
    }
    printf(")\n");
}
int main(void)
{
    node *n, *head = NULL;
    for (int i = 0; i < 2; i++) {
        n = malloc(sizeof(*n));
        n->val = 21 - i;
        n->next = head;
        head = n;
    }
    print_list(head);
    n = head; // Explicit cleanup
    while (n != NULL) {
        node *p = n->next;
        free(n);
        n = p;
    }
    return 0;
}
```



[redacted]

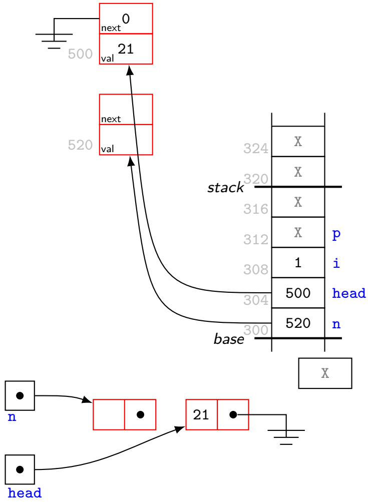
Lista 2: Typad payload, dynamiska noder

```c
typedef struct node {
    int val; // typed payload of fixed size inside struct
    struct node *next;
} node;
void print_list(const node *p)
{
    printf("( ");
    while (p != NULL) {
        printf("%d ", p->val); // we know the type of val
        p = p->next;
    }
    printf(")\n");
}
int main(void)
{
    node *n, *head = NULL;
    for (int i = 0; i < 2; i++) {
        n = malloc(sizeof(*n));
        n->val = 21 - i;
        n->next = head;
        head = n;
    }
    print_list(head);
    n = head; // Explicit cleanup
    while (n != NULL) {
        node *p = n->next;
        free(n);
        n = p;
    }
    return 0;
}
```



Lista 2: Typad payload, dynamiska noder

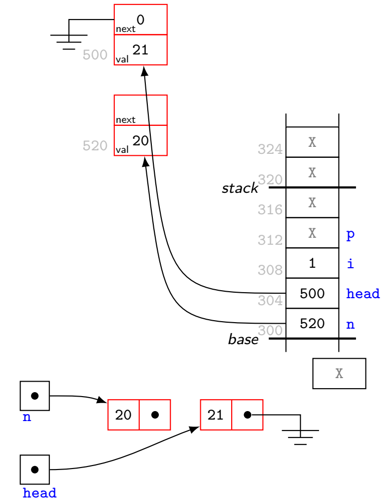
```c
typedef struct node {
    int val; // typed payload of fixed size inside struct
    struct node *next;
} node;
void print_list(const node *p)
{
    printf("( ");
    while (p != NULL) {
        printf("%d ", p->val); // we know the type of val
        p = p->next;
    }
    printf(")\n");
}
int main(void)
{
    node *n, *head = NULL;
    for (int i = 0; i < 2; i++) {
        n = malloc(sizeof(*n));
        n->val = 21 - i;
        n->next = head;
        head = n;
    }
    print_list(head);
    n = head; // Explicit cleanup
    while (n != NULL) {
        node *p = n->next;
        free(n);
        n = p;
    }
    return 0;
}
```



[redacted]

Lista 2: Typad payload, dynamiska noder

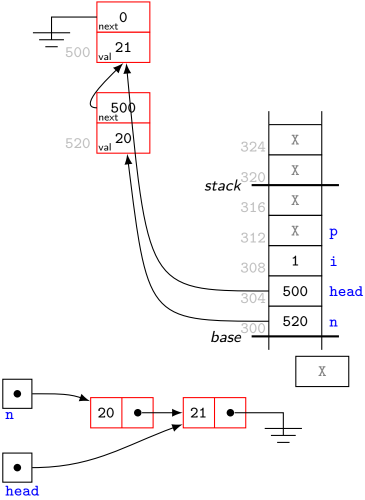
```c
typedef struct node {
    int val; // typed payload of fixed size inside struct
    struct node *next;
} node;
void print_list(const node *p)
{
    printf("( ");
    while (p != NULL) {
        printf("%d ", p->val); // we know the type of val
        p = p->next;
    }
    printf(")\n");
}
int main(void)
{
    node *n, *head = NULL;
    for (int i = 0; i < 2; i++) {
        n = malloc(sizeof(*n));
        n->val = 21 - i;
        n->next = head;
        head = n;
    }
    print_list(head);
    n = head; // Explicit cleanup
    while (n != NULL) {
        node *p = n->next;
        free(n);
        n = p;
    }
    return 0;
}
```



[redacted]

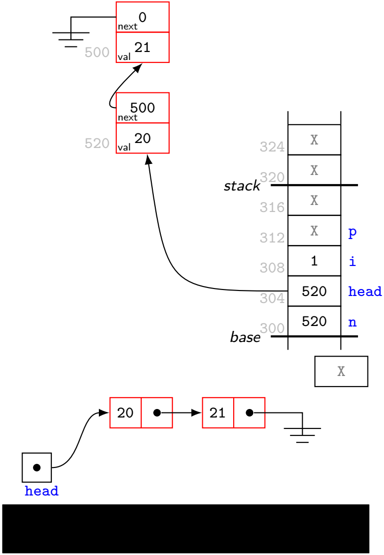
Lista 2: Typad payload, dynamiska noder

```c
typedef struct node {
    int val; // typed payload of fixed size inside struct
    struct node *next;
} node;
void print_list(const node *p)
{
    printf("( ");
    while (p != NULL) {
        printf("%d ", p->val); // we know the type of val
        p = p->next;
    }
    printf(")\n");
}
int main(void)
{
    node *n, *head = NULL;
    for (int i = 0; i < 2; i++) {
        n = malloc(sizeof(*n));
        n->val = 21 - i;
        n->next = head;
        head = n;
    }
    print_list(head);
    n = head; // Explicit cleanup
    while (n != NULL) {
        node *p = n->next;
        free(n);
        n = p;
    }
    return 0;
}
```



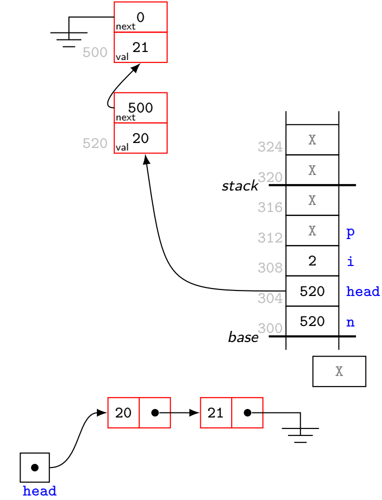
Lista 2: Typad payload, dynamiska noder

```c
typedef struct node {
    int val; // typed payload of fixed size inside struct
    struct node *next;
} node;
void print_list(const node *p)
{
    printf("( ");
    while (p != NULL) {
        printf("%d ", p->val); // we know the type of val
        p = p->next;
    }
    printf(")\n");
}
int main(void)
{
    node *n, *head = NULL;
    for (int i = 0; i < 2; i++) {
        n = malloc(sizeof(*n));
        n->val = 21 - i;
        n->next = head;
        head = n;
    }
    print_list(head);
    n = head; // Explicit cleanup
    while (n != NULL) {
        node *p = n->next;
        free(n);
        n = p;
    }
    return 0;
}
```



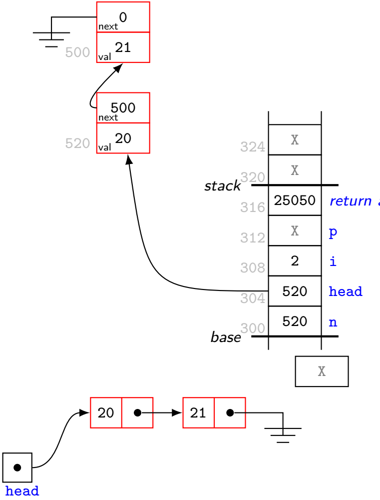
Lista 2: Typad payload, dynamiska noder

```c
typedef struct node {
    int val; // typed payload of fixed size inside struct
    struct node *next;
} node;
void print_list(const node *p)
{
    printf("( ");
    while (p != NULL) {
        printf("%d ", p->val); // we know the type of val
        p = p->next;
    }
    printf(")\n");
}
int main(void)
{
    node *n, *head = NULL;
    for (int i = 0; i < 2; i++) {
        n = malloc(sizeof(*n));
        n->val = 21 - i;
        n->next = head;
        head = n;
    }
    print_list(head);
    n = head; // Explicit cleanup
    while (n != NULL) {
        node *p = n->next;
        free(n);
        n = p;
    }
    return 0;
}
```



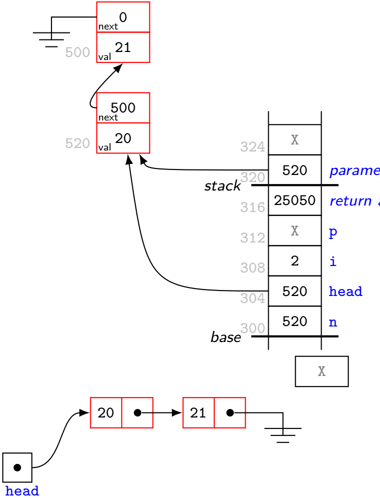
Lista 2: Typad payload, dynamiska noder

```c
typedef struct node {
    int val; // typed payload of fixed size inside struct
    struct node *next;
} node;
void print_list(const node *p)
{
    printf("( ");
    while (p != NULL) {
        printf("%d ", p->val); // we know the type of val
        p = p->next;
    }
    printf(")\n");
}
int main(void)
{
    node *n, *head = NULL;
    for (int i = 0; i < 2; i++) {
        n = malloc(sizeof(*n));
        n->val = 21 - i;
        n->next = head;
        head = n;
    }
    print_list(head);
    n = head; // Explicit cleanup
    while (n != NULL) {
        node *p = n->next;
        free(n);
        n = p;
    }
    return 0;
}
```



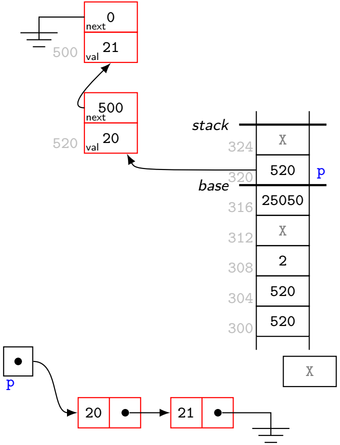
Lista 2: Typad payload, dynamiska noder

```c
typedef struct node {
    int val; // typed payload of fixed size inside struct
    struct node *next;
} node;
void print_list(const node *p)
{
    printf("( ");
    while (p != NULL) {
        printf("%d ", p->val); // we know the type of val
        p = p->next;
    }
    printf(")\n");
}
int main(void)
{
    node *n, *head = NULL;
    for (int i = 0; i < 2; i++) {
        n = malloc(sizeof(*n));
        n->val = 21 - i;
        n->next = head;
        head = n;
    }
    print_list(head);
    n = head; // Explicit cleanup
    while (n != NULL) {
        node *p = n->next;
        free(n);
        n = p;
    }
    return 0;
}
```



Lista 2: Typad payload, dynamiska noder

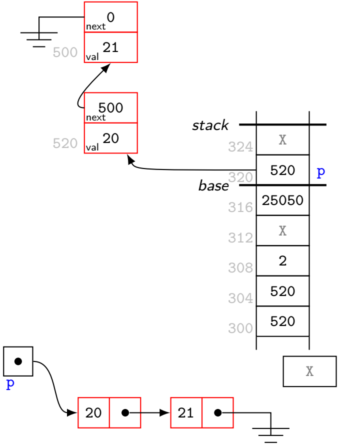
```c
typedef struct node {
    int val; // typed payload of fixed size inside struct
    struct node *next;
} node;
void print_list(const node *p)
{
    printf("( ");
    while (p != NULL) {
        printf("%d ", p->val); // we know the type of val
        p = p->next;
    }
    printf(")\n");
}
int main(void)
{
    node *n, *head = NULL;
    for (int i = 0; i < 2; i++) {
        n = malloc(sizeof(*n));
        n->val = 21 - i;
        n->next = head;
        head = n;
    }
    print_list(head);
    n = head; // Explicit cleanup
    while (n != NULL) {
        node *p = n->next;
        free(n);
        n = p;
    }
    return 0;
}
```



[redacted]

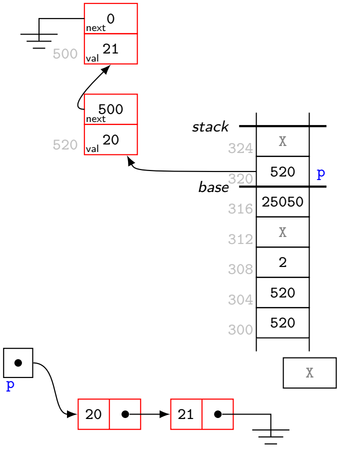
Lista 2: Typad payload, dynamiska noder

```c
typedef struct node {
    int val; // typed payload of fixed size inside struct
    struct node *next;
} node;
void print_list(const node *p)
{
    printf("( ");
    while (p != NULL) {
        printf("%d ", p->val); // we know the type of val
        p = p->next;
    }
    printf(")\n");
}
int main(void)
{
    node *n, *head = NULL;
    for (int i = 0; i < 2; i++) {
        n = malloc(sizeof(*n));
        n->val = 21 - i;
        n->next = head;
        head = n;
    }
    print_list(head);
    n = head; // Explicit cleanup
    while (n != NULL) {
        node *p = n->next;
        free(n);
        n = p;
    }
    return 0;
}
```



Lista 2: Typad payload, dynamiska noder

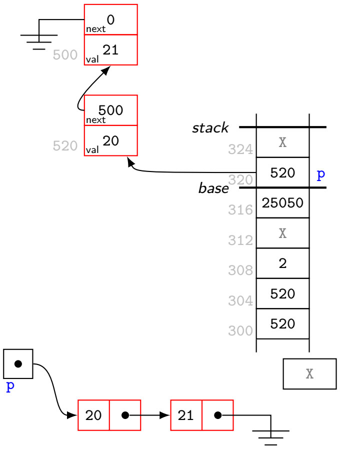
```c
typedef struct node {
    int val; // typed payload of fixed size inside struct
    struct node *next;
} node;
void print_list(const node *p)
{
    printf("( ");
    while (p != NULL) {
        printf("%d ", p->val); // we know the type of val
        p = p->next;
    }
    printf(")\n");
}
int main(void)
{
    node *n, *head = NULL;
    for (int i = 0; i < 2; i++) {
        n = malloc(sizeof(*n));
        n->val = 21 - i;
        n->next = head;
        head = n;
    }
    print_list(head);
    n = head; // Explicit cleanup
    while (n != NULL) {
        node *p = n->next;
        free(n);
        n = p;
    }
    return 0;
}
```




```
(
```

Lista 2: Typad payload, dynamiska noder

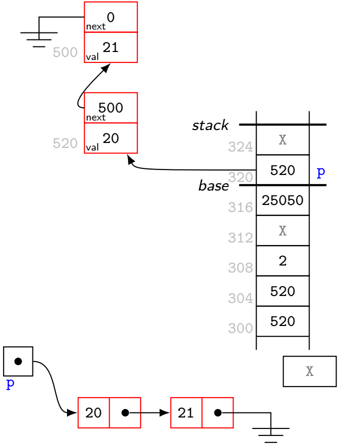
```c
typedef struct node {
    int val; // typed payload of fixed size inside struct
    struct node *next;
} node;
void print_list(const node *p)
{
    printf("( ");
    while (p != NULL) {
        printf("%d ", p->val); // we know the type of val
        p = p->next;
    }
    printf(")\n");
}
int main(void)
{
    node *n, *head = NULL;
    for (int i = 0; i < 2; i++) {
        n = malloc(sizeof(*n));
        n->val = 21 - i;
        n->next = head;
        head = n;
    }
    print_list(head);
    n = head; // Explicit cleanup
    while (n != NULL) {
        node *p = n->next;
        free(n);
        n = p;
    }
    return 0;
}
```




```
(
```

Lista 2: Typad payload, dynamiska noder

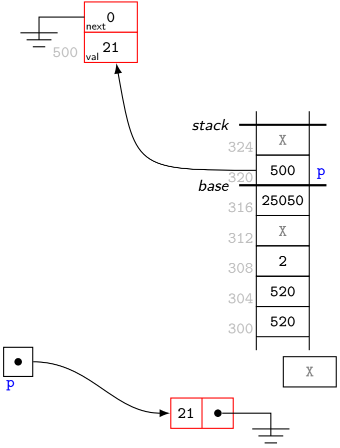
```
typedef struct node {
    int val; // typed payload of fixed size inside struct
    struct node *next;
} node;
void print_list(const node *p)
{
    printf("( ");
    while (p != NULL) {
        printf("%d ", p->val); // we know the type of val
        p = p->next;
    }
    printf(")\n");
}
int main(void)
{
    node *n, *head = NULL;
    for (int i = 0; i < 2; i++) {
        n = malloc(sizeof(*n));
        n->val = 21 - i;
        n->next = head;
        head = n;
    }
    print_list(head);
    n = head; // Explicit cleanup
    while (n != NULL) {
        node *p = n->next;
        free(n);
        n = p;
    }
    return 0;
}
```




```
( 20
```

Lista 2: Typad payload, dynamiska noder

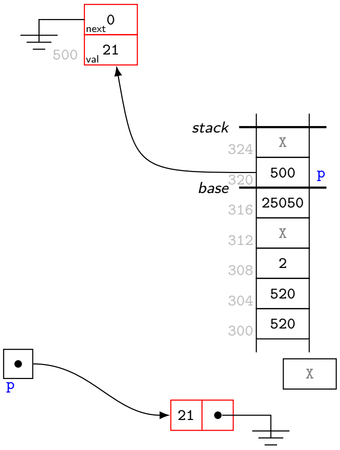
```c
typedef struct node {
    int val; // typed payload of fixed size inside struct
    struct node *next;
} node;
void print_list(const node *p)
{
    printf("( ");
    while (p != NULL) {
        printf("%d ", p->val); // we know the type of val
        p = p->next;
    }
    printf(")\n");
}
int main(void)
{
    node *n, *head = NULL;
    for (int i = 0; i < 2; i++) {
        n = malloc(sizeof(*n));
        n->val = 21 - i;
        n->next = head;
        head = n;
    }
    print_list(head);
    n = head; // Explicit cleanup
    while (n != NULL) {
        node *p = n->next;
        free(n);
        n = p;
    }
    return 0;
}
```



(20

Lista 2: Typad payload, dynamiska noder

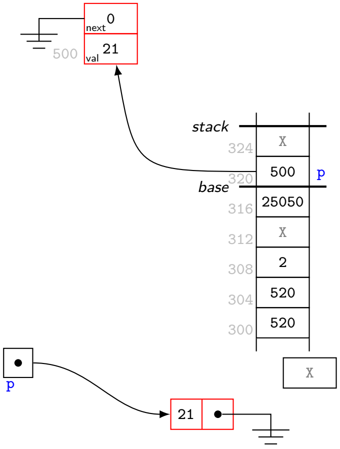
```c
typedef struct node {
    int val; // typed payload of fixed size inside struct
    struct node *next;
} node;
void print_list(const node *p)
{
    printf("( ");
    while (p != NULL) {
        printf("%d ", p->val); // we know the type of val
        p = p->next;
    }
    printf(")\n");
}
int main(void)
{
    node *n, *head = NULL;
    for (int i = 0; i < 2; i++) {
        n = malloc(sizeof(*n));
        n->val = 21 - i;
        n->next = head;
        head = n;
    }
    print_list(head);
    n = head; // Explicit cleanup
    while (n != NULL) {
        node *p = n->next;
        free(n);
        n = p;
    }
    return 0;
}
```




```
( 20
```

Lista 2: Typad payload, dynamiska noder

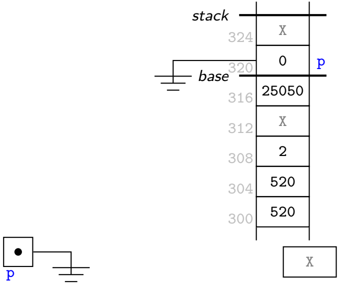
```c
typedef struct node {
    int val; // typed payload of fixed size inside struct
    struct node *next;
} node;
void print_list(const node *p)
{
    printf("( ");
    while (p != NULL) {
        printf("%d ", p->val); // we know the type of val
        p = p->next;
    }
    printf(")\n");
}
int main(void)
{
    node *n, *head = NULL;
    for (int i = 0; i < 2; i++) {
        n = malloc(sizeof(*n));
        n->val = 21 - i;
        n->next = head;
        head = n;
    }
    print_list(head);
    n = head; // Explicit cleanup
    while (n != NULL) {
        node *p = n->next;
        free(n);
        n = p;
    }
    return 0;
}
```




```
( 20 21
```

Lista 2: Typad payload, dynamiska noder

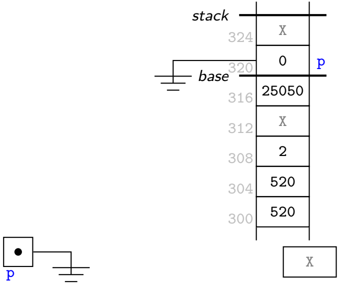
```c
typedef struct node {
    int val; // typed payload of fixed size inside struct
    struct node *next;
} node;
void print_list(const node *p)
{
    printf("( ");
    while (p != NULL) {
        printf("%d ", p->val); // we know the type of val
        p = p->next;
    }
    printf(")\n");
}
int main(void)
{
    node *n, *head = NULL;
    for (int i = 0; i < 2; i++) {
        n = malloc(sizeof(*n));
        n->val = 21 - i;
        n->next = head;
        head = n;
    }
    print_list(head);
    n = head; // Explicit cleanup
    while (n != NULL) {
        node *p = n->next;
        free(n);
        n = p;
    }
    return 0;
}
```



(20 21

Lista 2: Typad payload, dynamiska noder

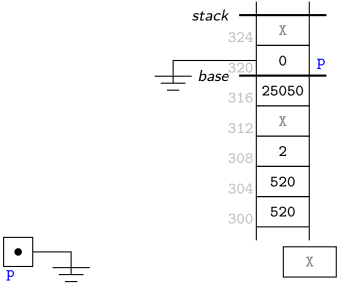
```c
typedef struct node {
    int val; // typed payload of fixed size inside struct
    struct node *next;
} node;
void print_list(const node *p)
{
    printf("( ");
    while (p != NULL) {
        printf("%d ", p->val); // we know the type of val
        p = p->next;
    }
    printf(")\n");
}
int main(void)
{
    node *n, *head = NULL;
    for (int i = 0; i < 2; i++) {
        n = malloc(sizeof(*n));
        n->val = 21 - i;
        n->next = head;
        head = n;
    }
    print_list(head);
    n = head; // Explicit cleanup
    while (n != NULL) {
        node *p = n->next;
        free(n);
        n = p;
    }
    return 0;
}
```




```
( 20 21
```

Lista 2: Typad payload, dynamiska noder

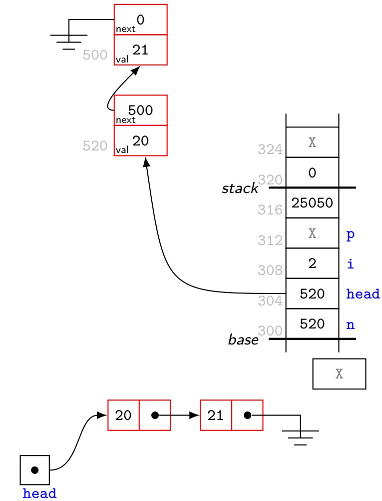
```c
typedef struct node {
    int val; // typed payload of fixed size inside struct
    struct node *next;
} node;
void print_list(const node *p)
{
    printf("( ");
    while (p != NULL) {
        printf("%d ", p->val); // we know the type of val
        p = p->next;
    }
    printf(")\n");
}
int main(void)
{
    node *n, *head = NULL;
    for (int i = 0; i < 2; i++) {
        n = malloc(sizeof(*n));
        n->val = 21 - i;
        n->next = head;
        head = n;
    }
    print_list(head);
    n = head; // Explicit cleanup
    while (n != NULL) {
        node *p = n->next;
        free(n);
        n = p;
    }
    return 0;
}
```




```
( 20 21 )
```

Lista 2: Typad payload, dynamiska noder

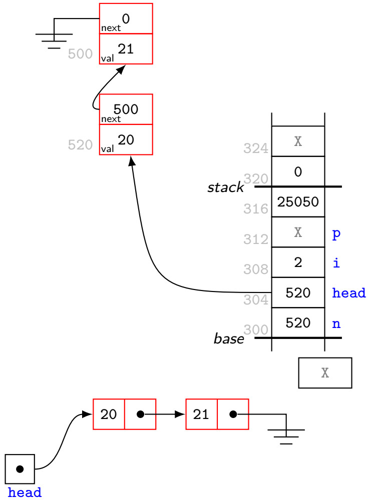
```c
typedef struct node {
    int val; // typed payload of fixed size inside struct
    struct node *next;
} node;
void print_list(const node *p)
{
    printf("( ");
    while (p != NULL) {
        printf("%d ", p->val); // we know the type of val
        p = p->next;
    }
    printf(")\n");
}
int main(void)
{
    node *n, *head = NULL;
    for (int i = 0; i < 2; i++) {
        n = malloc(sizeof(*n));
        n->val = 21 - i;
        n->next = head;
        head = n;
    }
    print_list(head);
    n = head; // Explicit cleanup
    while (n != NULL) {
        node *p = n->next;
        free(n);
        n = p;
    }
    return 0;
}
```




```
( 20 21 )
```

Lista 2: Typad payload, dynamiska noder

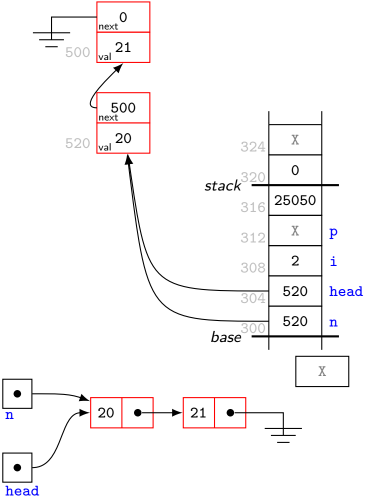
```c
typedef struct node {
    int val; // typed payload of fixed size inside struct
    struct node *next;
} node;
void print_list(const node *p)
{
    printf("( ");
    while (p != NULL) {
        printf("%d ", p->val); // we know the type of val
        p = p->next;
    }
    printf(")\n");
}
int main(void)
{
    node *n, *head = NULL;
    for (int i = 0; i < 2; i++) {
        n = malloc(sizeof(*n));
        n->val = 21 - i;
        n->next = head;
        head = n;
    }
    print_list(head);
    n = head; // Explicit cleanup
    while (n != NULL) {
        node *p = n->next;
        free(n);
        n = p;
    }
    return 0;
}
```



(20 21)

Lista 2: Typad payload, dynamiska noder

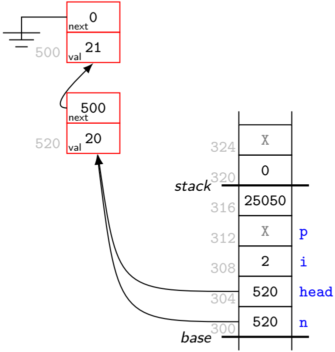
```c
typedef struct node {
    int val; // typed payload of fixed size inside struct
    struct node *next;
} node;
void print_list(const node *p)
{
    printf("( ");
    while (p != NULL) {
        printf("%d ", p->val); // we know the type of val
        p = p->next;
    }
    printf(")\n");
}
int main(void)
{
    node *n, *head = NULL;
    for (int i = 0; i < 2; i++) {
        n = malloc(sizeof(*n));
        n->val = 21 - i;
        n->next = head;
        head = n;
    }
    print_list(head);
    n = head; // Explicit cleanup
    while (n != NULL) {
        node *p = n->next;
        free(n);
        n = p;
    }
    return 0;
}
```

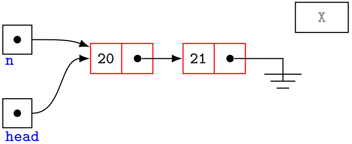


(20 21)

Lista 2: Typad payload, dynamiska noder

```c
typedef struct node {
    int val; // typed payload of fixed size inside struct
    struct node *next;
} node;
void print_list(const node *p)
{
    printf("( ");
    while (p != NULL) {
        printf("%d ", p->val); // we know the type of val
        p = p->next;
    }
    printf(")\n");
}
int main(void)
{
    node *n, *head = NULL;
    for (int i = 0; i < 2; i++) {
        n = malloc(sizeof(*n));
        n->val = 21 - i;
        n->next = head;
        head = n;
    }
    print_list(head);
    n = head; // Explicit cleanup
    while (n != NULL) {
        node *p = n->next;
        free(n);
        n = p;
    }
    return 0;
}
```

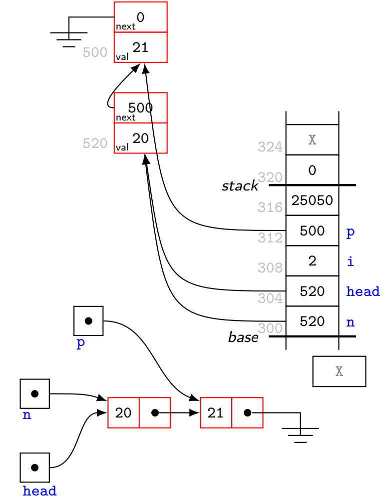





```
( 20 21 )
```

Lista 2: Typad payload, dynamiska noder

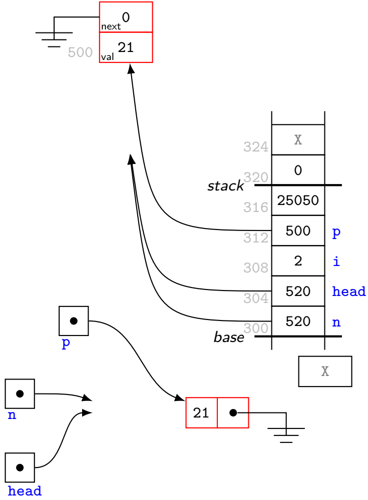
```c
typedef struct node {
    int val; // typed payload of fixed size inside struct
    struct node *next;
} node;
void print_list(const node *p)
{
    printf("( ");
    while (p != NULL) {
        printf("%d ", p->val); // we know the type of val
        p = p->next;
    }
    printf(")\n");
}
int main(void)
{
    node *n, *head = NULL;
    for (int i = 0; i < 2; i++) {
        n = malloc(sizeof(*n));
        n->val = 21 - i;
        n->next = head;
        head = n;
    }
    print_list(head);
    n = head; // Explicit cleanup
    while (n != NULL) {
        node *p = n->next;
        free(n);
        n = p;
    }
    return 0;
}
```



(20 21)

Lista 2: Typad payload, dynamiska noder

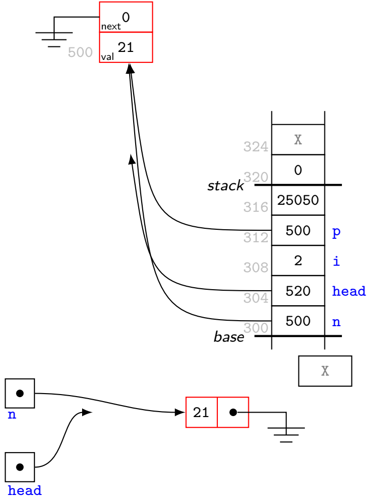
```c
typedef struct node {
    int val; // typed payload of fixed size inside struct
    struct node *next;
} node;
void print_list(const node *p)
{
    printf("( ");
    while (p != NULL) {
        printf("%d ", p->val); // we know the type of val
        p = p->next;
    }
    printf(")\n");
}
int main(void)
{
    node *n, *head = NULL;
    for (int i = 0; i < 2; i++) {
        n = malloc(sizeof(*n));
        n->val = 21 - i;
        n->next = head;
        head = n;
    }
    print_list(head);
    n = head; // Explicit cleanup
    while (n != NULL) {
        node *p = n->next;
        free(n);
        n = p;
    }
    return 0;
}
```



(20 21)

Lista 2: Typad payload, dynamiska noder

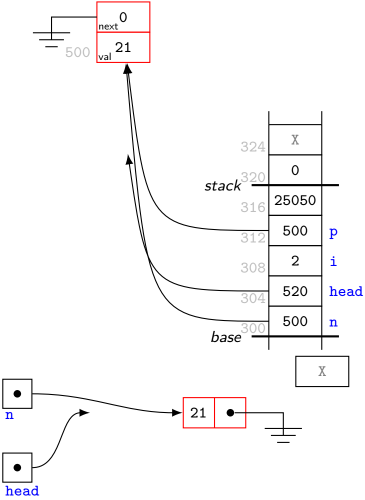
```c
typedef struct node {
    int val; // typed payload of fixed size inside struct
    struct node *next;
} node;
void print_list(const node *p)
{
    printf("( ");
    while (p != NULL) {
        printf("%d ", p->val); // we know the type of val
        p = p->next;
    }
    printf(")\n");
}
int main(void)
{
    node *n, *head = NULL;
    for (int i = 0; i < 2; i++) {
        n = malloc(sizeof(*n));
        n->val = 21 - i;
        n->next = head;
        head = n;
    }
    print_list(head);
    n = head; // Explicit cleanup
    while (n != NULL) {
        node *p = n->next;
        free(n);
        n = p;
    }
    return 0;
}
```



(20 21)

Lista 2: Typad payload, dynamiska noder

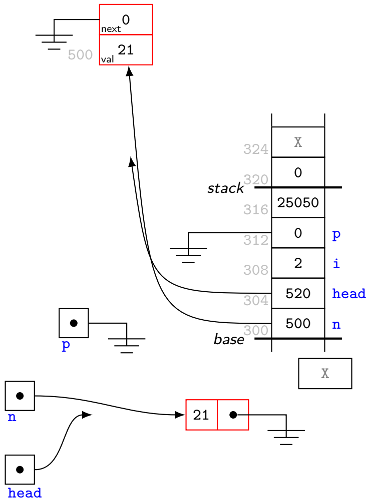
```c
typedef struct node {
    int val; // typed payload of fixed size inside struct
    struct node *next;
} node;
void print_list(const node *p)
{
    printf("( ");
    while (p != NULL) {
        printf("%d ", p->val); // we know the type of val
        p = p->next;
    }
    printf(")\n");
}
int main(void)
{
    node *n, *head = NULL;
    for (int i = 0; i < 2; i++) {
        n = malloc(sizeof(*n));
        n->val = 21 - i;
        n->next = head;
        head = n;
    }
    print_list(head);
    n = head; // Explicit cleanup
    while (n != NULL) {
        node *p = n->next;
        free(n);
        n = p;
    }
    return 0;
}
```




```
( 20 21 )
```

Lista 2: Typad payload, dynamiska noder

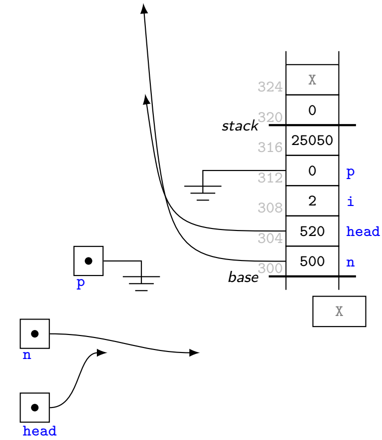
```c
typedef struct node {
    int val; // typed payload of fixed size inside struct
    struct node *next;
} node;
void print_list(const node *p)
{
    printf("( ");
    while (p != NULL) {
        printf("%d ", p->val); // we know the type of val
        p = p->next;
    }
    printf(")\n");
}
int main(void)
{
    node *n, *head = NULL;
    for (int i = 0; i < 2; i++) {
        n = malloc(sizeof(*n));
        n->val = 21 - i;
        n->next = head;
        head = n;
    }
    print_list(head);
    n = head; // Explicit cleanup
    while (n != NULL) {
        node *p = n->next;
        free(n);
        n = p;
    }
    return 0;
}
```




```
( 20 21 )
```

Lista 2: Typad payload, dynamiska noder

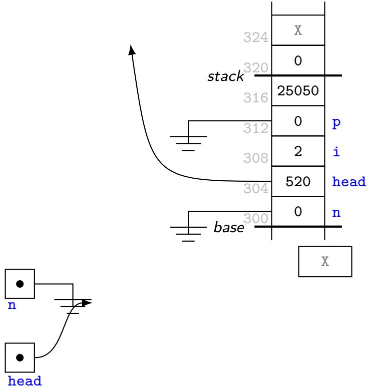
```c
typedef struct node {
    int val; // typed payload of fixed size inside struct
    struct node *next;
} node;
void print_list(const node *p)
{
    printf("( ");
    while (p != NULL) {
        printf("%d ", p->val); // we know the type of val
        p = p->next;
    }
    printf(")\n");
}
int main(void)
{
    node *n, *head = NULL;
    for (int i = 0; i < 2; i++) {
        n = malloc(sizeof(*n));
        n->val = 21 - i;
        n->next = head;
        head = n;
    }
    print_list(head);
    n = head; // Explicit cleanup
    while (n != NULL) {
        node *p = n->next;
        free(n);
        n = p;
    }
    return 0;
}
```




```
( 20 21 )
```

Lista 2: Typad payload, dynamiska noder

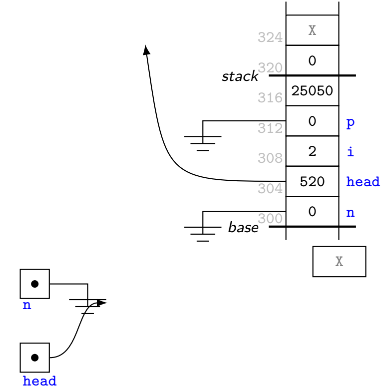
```c
typedef struct node {
    int val; // typed payload of fixed size inside struct
    struct node *next;
} node;
void print_list(const node *p)
{
    printf("( ");
    while (p != NULL) {
        printf("%d ", p->val); // we know the type of val
        p = p->next;
    }
    printf(")\n");
}
int main(void)
{
    node *n, *head = NULL;
    for (int i = 0; i < 2; i++) {
        n = malloc(sizeof(*n));
        n->val = 21 - i;
        n->next = head;
        head = n;
    }
    print_list(head);
    n = head; // Explicit cleanup
    while (n != NULL) {
        node *p = n->next;
        free(n);
        n = p;
    }
    return 0;
}
```




```
( 20 21 )
```

Lista 2: Typad payload, dynamiska noder

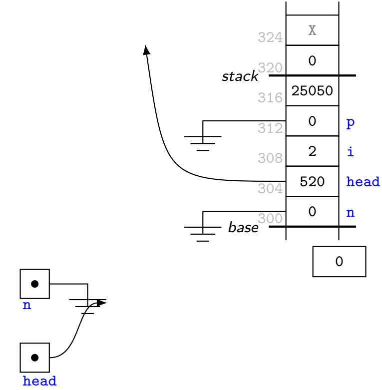
```c
typedef struct node {
    int val; // typed payload of fixed size inside struct
    struct node *next;
} node;
void print_list(const node *p)
{
    printf("( ");
    while (p != NULL) {
        printf("%d ", p->val); // we know the type of val
        p = p->next;
    }
    printf(")\n");
}
int main(void)
{
    node *n, *head = NULL;
    for (int i = 0; i < 2; i++) {
        n = malloc(sizeof(*n));
        n->val = 21 - i;
        n->next = head;
        head = n;
    }
    print_list(head);
    n = head; // Explicit cleanup
    while (n != NULL) {
        node *p = n->next;
        free(n);
        n = p;
    }
    return 0;
}
```




```
( 20 21 )
```

Lista 2: Typad payload, dynamiska noder

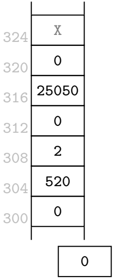
```c
typedef struct node {
    int val; // typed payload of fixed size inside struct
    struct node *next;
} node;
void print_list(const node *p)
{
    printf("( ");
    while (p != NULL) {
        printf("%d ", p->val); // we know the type of val
        p = p->next;
    }
    printf(")\n");
}
int main(void)
{
    node *n, *head = NULL;
    for (int i = 0; i < 2; i++) {
        n = malloc(sizeof(*n));
        n->val = 21 - i;
        n->next = head;
        head = n;
    }
    print_list(head);
    n = head; // Explicit cleanup
    while (n != NULL) {
        node *p = n->next;
        free(n);
        n = p;
    }
    return 0;
}
```




```
( 20 21 )
```

Lista 2: Typad payload, dynamiska noder

```c
typedef struct node {
    int val; // typed payload of fixed size inside struct
    struct node *next;
} node;
void print_list(const node *p)
{
    printf("( ");
    while (p != NULL) {
        printf("%d ", p->val); // we know the type of val
        p = p->next;
    }
    printf(")\n");
}
int main(void)
{
    node *n, *head = NULL;
    for (int i = 0; i < 2; i++) {
        n = malloc(sizeof(*n));
        n->val = 21 - i;
        n->next = head;
        head = n;
    }
    print_list(head);
    n = head; // Explicit cleanup
    while (n != NULL) {
        node *p = n->next;
        free(n);
        n = p;
    }
    return 0;
}
```



(20 21)

Generiska datatyper i C

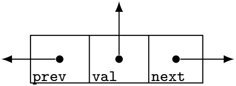
Generiska datatyper

- De första list-exemplen lagrar en variabel av en bestämd typ (`int`) inuti elementen
- Vill vi ha friheten att lagra vilken typ som helst — och vilken storlek som helst — kan vi använda `void`-pekare
- Kodbasen använder följande `struct` i `list.c`:

code/list.c

```
typedef struct cell {
    struct cell *next;
    struct cell *prev;
    void *val;
} cell;
```

- Denna `struct` motsvarar en 2-cell, där värdet också är en länk



`void`-pekare (1)

- En `void`-pekare är en pekare utan information om typen
  - En `void`-pekare innehåller bara en adress
  - Dereferering av en `void`-pekare är inte tillåten (kompilatorfel)
- Kompilatorn tillåter att man gör tilldelningar mellan `void`-pekare och typade pekare och vice versa

```
int i=4;
int *p=&i;
void *r=p;
int *q=r;
```

- Vill man vara tydlig kan man använda en s.k. *type cast*

```
int *q=(int *)r;
```

`void`-pekare (2)

- Det är vanligt att **`void`**-pekare används av funktioner som inte behöver veta vilken typ som pekaren har
- En funktion som behöver tolka datat måste översätta **`void`**-pekaren till en typad pekare
- Notera att det finns inget skydd i språket mot att adressen tilldelas en pekare av fel sort

Hur kodbasen hanterar abstraktion (1)

- Kodbasen utnyttjar två tekniker för att uppnå ett visst mått av abstraktion
  1. `void`-pekare
  2. Pekare till anonyma `struct`-ar
- Header-filen `list.h` deklarerar två typer:
  - `list` som är en `struct` med okänt innehåll
  - `list_pos` som är en pekare till `struct cell`, återigen med okänt innehåll

code/list.h

```
// List type.
typedef struct list list;

// List position type.
typedef struct cell *list_pos;
```

Hur kodbasen hanterar abstraktion (2)

- Huvudprogrammet inkluderar `list.h`

code/list-with-freehandler.c

```c
#include <list.h>
```

och kan då använda variabler av typen `list *` och `list_pos` (`struct cell *`) utan att känna till innehållet i respektive `struct`

code/list-with-freehandler.c

```c
int main(void)
{
    // Create empty list, hand over responsibility to
    // deallocate payload using int_kill
    list *l = list_empty(int_kill);
    list_pos p = list_first(l);
    for (int i = 0; i < 2; i++) {
        int *v = int_create(20 + i);
        p = list_insert(l, v, p);
        p = list_next(l, p);
    }
    list_print(l, print_int);






    // Clean up the list, including payload
    list_kill(l);
    return 0;
}
```

Hur kodbasen hanterar abstraktion (3)

- Koden i `list.c` definierar innehållet i **struct**-arna
  - Notera att värdet som lagras är av typen **void** *

code/list.c

```
/*
 * The list elements are implemented as two-cells with forward and
 * backward links and a void * for the value. The list uses two border
 * cells at the start and end of the list.
 */
typedef struct cell {
    struct cell *next;
    struct cell *prev;
    void *val;
} cell;

struct list {
    cell *head;
    cell *tail;
    kill_function kill_func;
};
```

- Funktionerna definierade längre ner i `list.c` känner till och kan använda fälten i **struct**-arna

code/list.c

```
list_pos list_next(const list *l, const list_pos p)
{
    return p->next;
}
```

Hur kodbasen hanterar abstraktion (4)

- `Insert` vet t.ex. hur den ska skapa ett nytt element, sätta dess värde och länka in det i listan

code/list.c

```c
list_pos list_insert(list *l, void *v, const list_pos p)
{
    // Allocate memory for a new cell.
    list_pos e = malloc(sizeof(cell));

    // Store the value.
    e->val = v;
    // Add links to/from the new cell.
    e->next = p;
    e->prev = p->prev;
    e->prev->next = e;
    p->prev = e;

    // Return the position of the new cell.
    return e;
}
```

- `Insert` känner dock inte till vad som lagras i listan, bara adressen till elementet

Hur kodbasen hanterar abstraktion (5)

- Vi har alltså en sorts tvåvägs-abstraktion:
  - Listan känner till hur listan och dess element är uppbyggda, men inte vilken datatyp som lagras i den
  - Huvudprogrammet känner å andra sidan till vad som stoppas in i listan, men inget om hur listan är uppbyggd

code/list-with-freehandler.c

```c
int *int_create(int v)
{
    int *p = malloc(sizeof(*p)); // allocate payload
    *p = v; // assign value
    return p;
}
```

code/list-with-freehandler.c

```c
int main(void)
{
    // Create empty list, hand over responsibility to
    // deallocate payload using int_kill
    list *l = list_empty(int_kill);
    list_pos p = list_first(l);
    for (int i = 0; i < 2; i++) {
        int *v = int_create(20 + i);
        p = list_insert(l, v, p);
        p = list_next(l, p);
    }
    list_print(l, print_int);
```

Hur kodbasen hanterar abstraktion (6)

- Nästa exempel använder kodbasen och innehåller ett huvudprogram och tre hjälpfunktioner:

code/list-with-killfunction.c

```c
#include <stdio.h>
#include <stdlib.h>
#include <list.h>
void print_int(const void *p)
{
    const int *q = p;
    printf("%d ", *q);
}
int *int_create(int v)
{
    int *p = malloc(sizeof(*p)); // allocate payload
    *p = v; // assign value
    return p;
}
void int_kill(void *v)
{
    free(v); // deallocate payload
}
```

- Hjälpfunktionerna är:

  `int_create` Allokera dynamiskt minne som rymmer ett heltal, sätt värdet och returnera adressen

  `int_kill` Lämna tillbaka minnet som tillhör ett heltal

  `print_int` Tolka innehållet i minnet på en viss adress som ett heltal och skriv ut värdet

Hur kodbasen hanterar abstraktion (7)

- Huvudprogrammet ser ut så här:

code/list-with-killfunction.c

```c
int main(void)
{
    // Create empty list, hand over responsibility to
    // deallocate payload using int_kill
    list *l = list_empty(int_kill);
    list_pos p = list_first(l);
    for (int i = 0; i < 2; i++) {
        int *v = int_create(20 + i);
        p = list_insert(l, v, p);
        p = list_next(l, p);
    }
    list_print(l, print_int);






    // Clean up the list, including payload
    list_kill(l);
    return 0;
}
```

- Huvudprogrammet skapar en tom lista och lägger in två st heltal (20 och 21) skapade med `int_create`
  - (Bortse för stunden från parametern `int_kill`)

Hur kodbasen hanterar abstraktion (8)

- Därefter skrivs listan ut med `list_print`:

code/list-with-killfunction.c

```c
int main(void)
{
    // Create empty list, hand over responsibility to
    // deallocate payload using int_kill
    list *l = list_empty(int_kill);
    list_pos p = list_first(l);
    for (int i = 0; i < 2; i++) {
        int *v = int_create(20 + i);
        p = list_insert(l, v, p);
        p = list_next(l, p);
    }
    list_print(l, print_int);

```

- Funktionen `list_print` tillhör listan och vet hur man traverserar listan för att komma åt alla element
- Funktionen `list_print` vet däremot inte hur värdet som lagras i varje element ska tolkas och skrivas ut
  - Därför skickar vi med en pekare till funktionen `print_int`

    `print_int` Tolka innehållet i minnet på en viss adress som ett heltal och skriv ut värdet

Hur kodbasen hanterar abstraktion (9)

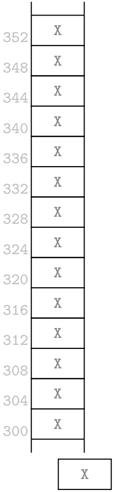
- Slutligen så anropas `list_kill` för att återlämna allt minne som listan använder:

code/list-with-killfunction.c

```c
int main(void)
{
    // Create empty list, hand over responsibility to
    // deallocate payload using int_kill
    list *l = list_empty(int_kill);
    list_pos p = list_first(l);
    for (int i = 0; i < 2; i++) {
        int *v = int_create(20 + i);
        p = list_insert(l, v, p);
        p = list_next(l, p);
    }
    list_print(l, print_int);

    // Clean up the list, including payload
    list_kill(l);
    return 0;
}
```

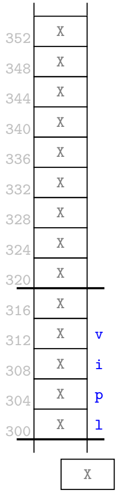
Exempel:

```c
int *int_create(int v)
{
    int *p = malloc(sizeof(*p)); // allocate payload
    *p = v; // assign value
    return p;
}
void int_kill(void *v)
{
    free(v); // deallocate payload
}
int main(void)
{
    // Create empty list, hand over responsibility to
    // deallocate payload using int_kill
    list *l = list_empty(int_kill);
    list_pos p = list_first(l);
    for (int i = 0; i < 2; i++) {
        int *v = int_create(20 + i);
        p = list_insert(l, v, p);
        p = list_next(l, p);
    }
    list_print(l, print_int);





    // Clean up the list, including payload
    list_kill(l);
    return 0;
}
```



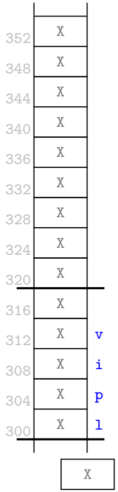
Exempel:

```c
int *int_create(int v)
{
    int *p = malloc(sizeof(*p)); // allocate payload
    *p = v; // assign value
    return p;
}
void int_kill(void *v)
{
    free(v); // deallocate payload
}
int main(void)
{
    // Create empty list, hand over responsibility to
    // deallocate payload using int_kill
    list *l = list_empty(int_kill);
    list_pos p = list_first(l);
    for (int i = 0; i < 2; i++) {
        int *v = int_create(20 + i);
        p = list_insert(l, v, p);
        p = list_next(l, p);
    }
    list_print(l, print_int);





    // Clean up the list, including payload
    list_kill(l);
    return 0;
}
```



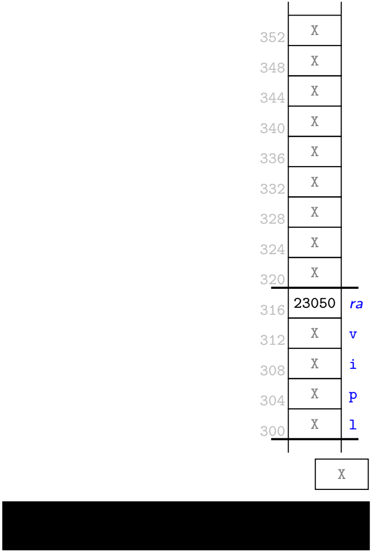
Exempel:

```c
int *int_create(int v)
{
    int *p = malloc(sizeof(*p)); // allocate payload
    *p = v; // assign value
    return p;
}
void int_kill(void *v)
{
    free(v); // deallocate payload
}
int main(void)
{
    // Create empty list, hand over responsibility to
    // deallocate payload using int_kill
    list *l = list_empty(int_kill);
    list_pos p = list_first(l);
    for (int i = 0; i < 2; i++) {
        int *v = int_create(20 + i);
        p = list_insert(l, v, p);
        p = list_next(l, p);
    }
    list_print(l, print_int);
    // Clean up the list, including payload
    list_kill(l);
    return 0;
}
```



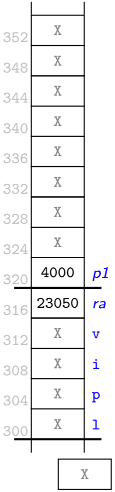
Exempel:

```c
int *int_create(int v)
{
    int *p = malloc(sizeof(*p)); // allocate payload
    *p = v; // assign value
    return p;
}
void int_kill(void *v)
{
    free(v); // deallocate payload
}
int main(void)
{
    // Create empty list, hand over responsibility to
    // deallocate payload using int_kill
    list *l = list_empty(int_kill);
    list_pos p = list_first(l);
    for (int i = 0; i < 2; i++) {
        int *v = int_create(20 + i);
        p = list_insert(l, v, p);
        p = list_next(l, p);
    }
    list_print(l, print_int);




    // Clean up the list, including payload
    list_kill(l);
    return 0;
}
```



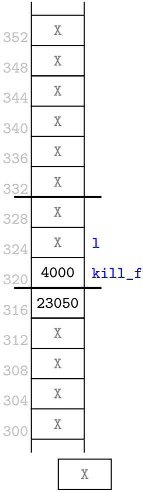
Exempel:

```c
int *int_create(int v)
{
    int *p = malloc(sizeof(*p)); // allocate payload
    *p = v; // assign value
    return p;
}
void int_kill(void *v)
{
    free(v); // deallocate payload
}
int main(void)
{
    // Create empty list, hand over responsibility to
    // deallocate payload using int_kill
    list *l = list_empty(int_kill);
    list_pos p = list_first(l);
    for (int i = 0; i < 2; i++) {
        int *v = int_create(20 + i);
        p = list_insert(l, v, p);
        p = list_next(l, p);
    }
    list_print(l, print_int);





    // Clean up the list, including payload
    list_kill(l);
    return 0;
}
```



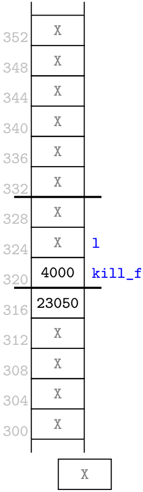
Exempel:

```c
/**
 * list_empty() - Create an empty list.
 * @kill_func: A pointer to a function (or NULL) to be ca
 *             de-allocate memory on remove/kill.
 *
 * Returns: A pointer to the new list.
 */
list *list_empty(kill_function kill_func)
{
    // Allocate memory for the list head.
    list *l = calloc(1, sizeof(list));

    // Allocate memory for the border cells.
    l->head = calloc(1, sizeof(cell));
    l->tail = calloc(1, sizeof(cell));

    // Set consistent links between border elements.
    l->head->next = l->tail;
    l->tail->prev = l->head;

    // Store the kill function.
    l->kill_func = kill_func;

    return l;
}
```



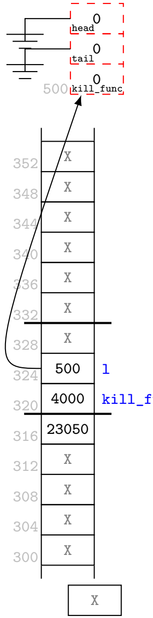
Exempel:

```c
/**
 * list_empty() - Create an empty list.
 * @kill_func: A pointer to a function (or NULL) to be ca
 *             de-allocate memory on remove/kill.
 *
 * Returns: A pointer to the new list.
 */
list *list_empty(kill_function kill_func)
{
    // Allocate memory for the list head.
    list *l = calloc(1, sizeof(list));

    // Allocate memory for the border cells.
    l->head = calloc(1, sizeof(cell));
    l->tail = calloc(1, sizeof(cell));

    // Set consistent links between border elements.
    l->head->next = l->tail;
    l->tail->prev = l->head;

    // Store the kill function.
    l->kill_func = kill_func;

    return l;
}
```



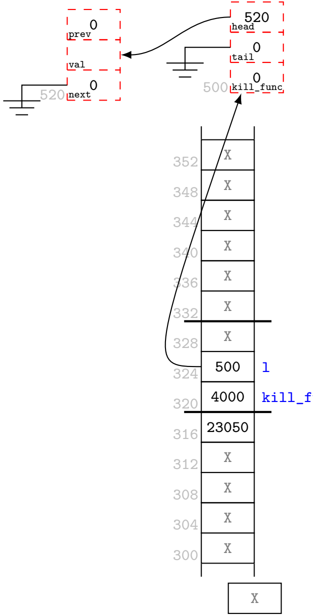
Exempel:

```c
/**
 * list_empty() - Create an empty list.
 * @kill_func: A pointer to a function (or NULL) to be ca
 *             de-allocate memory on remove/kill.
 *
 * Returns: A pointer to the new list.
 */
list *list_empty(kill_function kill_func)
{
    // Allocate memory for the list head.
    list *l = calloc(1, sizeof(list));

    // Allocate memory for the border cells.
    l->head = calloc(1, sizeof(cell));
    l->tail = calloc(1, sizeof(cell));

    // Set consistent links between border elements.
    l->head->next = l->tail;
    l->tail->prev = l->head;

    // Store the kill function.
    l->kill_func = kill_func;

    return l;
}
```



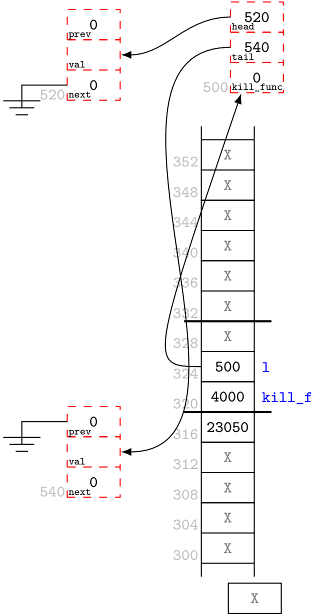
Exempel:

```c
/**
 * list_empty() - Create an empty list.
 * @kill_func: A pointer to a function (or NULL) to be ca
 *             de-allocate memory on remove/kill.
 *
 * Returns: A pointer to the new list.
 */
list *list_empty(kill_function kill_func)
{
    // Allocate memory for the list head.
    list *l = calloc(1, sizeof(list));

    // Allocate memory for the border cells.
    l->head = calloc(1, sizeof(cell));
    l->tail = calloc(1, sizeof(cell));

    // Set consistent links between border elements.
    l->head->next = l->tail;
    l->tail->prev = l->head;

    // Store the kill function.
    l->kill_func = kill_func;

    return l;
}
```



Exempel:

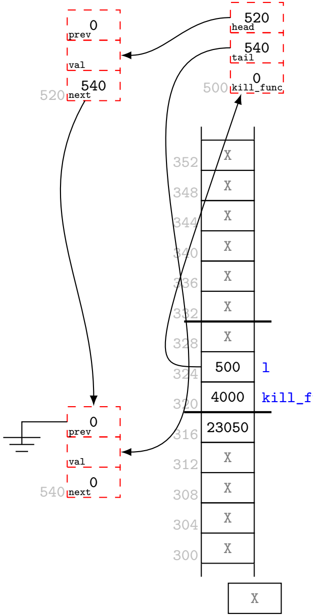
```c
/**
 * list_empty() - Create an empty list.
 * @kill_func: A pointer to a function (or NULL) to be ca
 *             de-allocate memory on remove/kill.
 *
 * Returns: A pointer to the new list.
 */
list *list_empty(kill_function kill_func)
{
    // Allocate memory for the list head.
    list *l = calloc(1, sizeof(list));

    // Allocate memory for the border cells.
    l->head = calloc(1, sizeof(cell));
    l->tail = calloc(1, sizeof(cell));

    // Set consistent links between border elements.
    l->head->next = l->tail;
    l->tail->prev = l->head;

    // Store the kill function.
    l->kill_func = kill_func;

    return l;
}
```



[redacted]

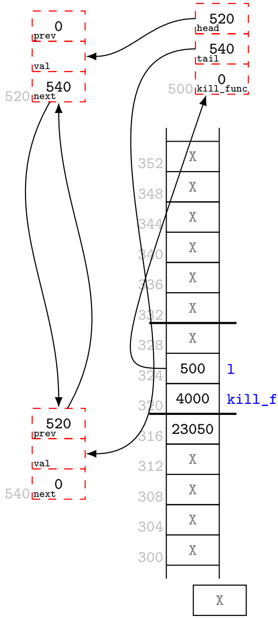
Exempel:

```c
/**
 * list_empty() - Create an empty list.
 * @kill_func: A pointer to a function (or NULL) to be ca
 *             de-allocate memory on remove/kill.
 *
 * Returns: A pointer to the new list.
 */
list *list_empty(kill_function kill_func)
{
    // Allocate memory for the list head.
    list *l = calloc(1, sizeof(list));

    // Allocate memory for the border cells.
    l->head = calloc(1, sizeof(cell));
    l->tail = calloc(1, sizeof(cell));

    // Set consistent links between border elements.
    l->head->next = l->tail;
    l->tail->prev = l->head;

    // Store the kill function.
    l->kill_func = kill_func;

    return l;
}
```



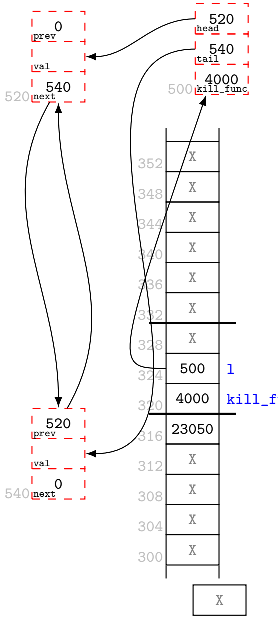
Exempel:

```c
/**
 * list_empty() - Create an empty list.
 * @kill_func: A pointer to a function (or NULL) to be ca
 *             de-allocate memory on remove/kill.
 *
 * Returns: A pointer to the new list.
 */
list *list_empty(kill_function kill_func)
{
    // Allocate memory for the list head.
    list *l = calloc(1, sizeof(list));

    // Allocate memory for the border cells.
    l->head = calloc(1, sizeof(cell));
    l->tail = calloc(1, sizeof(cell));

    // Set consistent links between border elements.
    l->head->next = l->tail;
    l->tail->prev = l->head;

    // Store the kill function.
    l->kill_func = kill_func;

    return l;
}
```



Exempel:

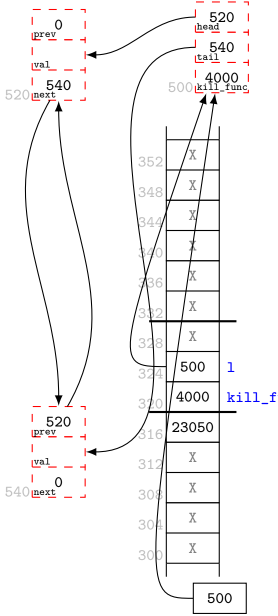
```c
/**
 * list_empty() - Create an empty list.
 * @kill_func: A pointer to a function (or NULL) to be ca
 *             de-allocate memory on remove/kill.
 *
 * Returns: A pointer to the new list.
 */
list *list_empty(kill_function kill_func)
{
    // Allocate memory for the list head.
    list *l = calloc(1, sizeof(list));

    // Allocate memory for the border cells.
    l->head = calloc(1, sizeof(cell));
    l->tail = calloc(1, sizeof(cell));

    // Set consistent links between border elements.
    l->head->next = l->tail;
    l->tail->prev = l->head;

    // Store the kill function.
    l->kill_func = kill_func;

    return l;
}
```



[redacted]

Exempel:

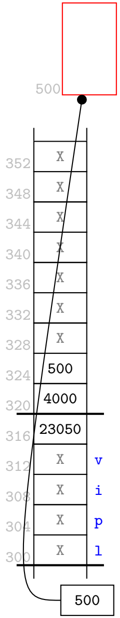
```c
/**
 * list_empty() - Create an empty list.
 * @kill_func: A pointer to a function (or NULL) to be ca
 *             de-allocate memory on remove/kill.
 *
 * Returns: A pointer to the new list.
 */
list *list_empty(kill_function kill_func)
{
    // Allocate memory for the list head.
    list *l = calloc(1, sizeof(list));

    // Allocate memory for the border cells.
    l->head = calloc(1, sizeof(cell));
    l->tail = calloc(1, sizeof(cell));

    // Set consistent links between border elements.
    l->head->next = l->tail;
    l->tail->prev = l->head;

    // Store the kill function.
    l->kill_func = kill_func;

    return l;
}
```



[redacted]

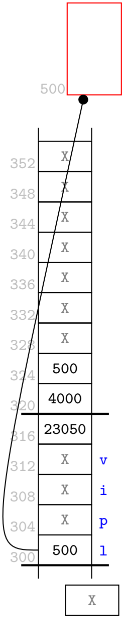
Exempel:

```c
int *int_create(int v)
{
    int *p = malloc(sizeof(*p)); // allocate payload
    *p = v; // assign value
    return p;
}
void int_kill(void *v)
{
    free(v); // deallocate payload
}
int main(void)
{
    // Create empty list, hand over responsibility to
    // deallocate payload using int_kill
    list *l = list_empty(int_kill);
    list_pos p = list_first(l);
    for (int i = 0; i < 2; i++) {
        int *v = int_create(20 + i);
        p = list_insert(l, v, p);
        p = list_next(l, p);
    }
    list_print(l, print_int);






    // Clean up the list, including payload
    list_kill(l);
    return 0;
}
```



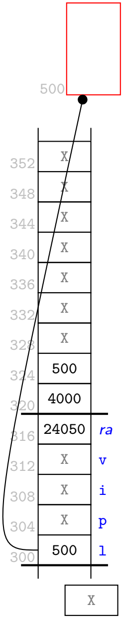
Exempel:

```c
int *int_create(int v)
{
    int *p = malloc(sizeof(*p)); // allocate payload
    *p = v; // assign value
    return p;
}
void int_kill(void *v)
{
    free(v); // deallocate payload
}
int main(void)
{
    // Create empty list, hand over responsibility to
    // deallocate payload using int_kill
    list *l = list_empty(int_kill);
    list_pos p = list_first(l);
    for (int i = 0; i < 2; i++) {
        int *v = int_create(20 + i);
        p = list_insert(l, v, p);
        p = list_next(l, p);
    }
    list_print(l, print_int);





    // Clean up the list, including payload
    list_kill(l);
    return 0;
}
```



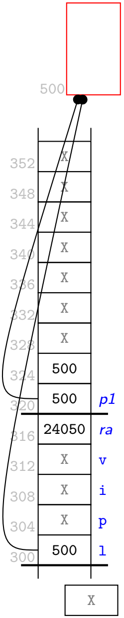
Exempel:

```c
int *int_create(int v)
{
    int *p = malloc(sizeof(*p)); // allocate payload
    *p = v; // assign value
    return p;
}
void int_kill(void *v)
{
    free(v); // deallocate payload
}
int main(void)
{
    // Create empty list, hand over responsibility to
    // deallocate payload using int_kill
    list *l = list_empty(int_kill);
    list_pos p = list_first(l);
    for (int i = 0; i < 2; i++) {
        int *v = int_create(20 + i);
        p = list_insert(l, v, p);
        p = list_next(l, p);
    }
    list_print(l, print_int);




    // Clean up the list, including payload
    list_kill(l);
    return 0;
}
```



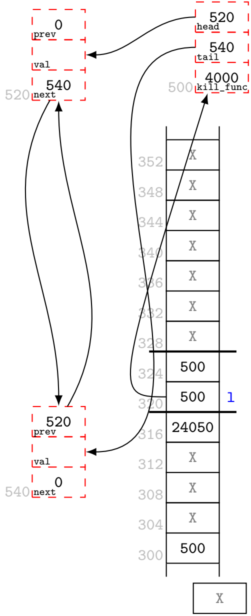
Exempel:

```c
int *int_create(int v)
{
    int *p = malloc(sizeof(*p)); // allocate payload
    *p = v; // assign value
    return p;
}
void int_kill(void *v)
{
    free(v); // deallocate payload
}
int main(void)
{
    // Create empty list, hand over responsibility to
    // deallocate payload using int_kill
    list *l = list_empty(int_kill);
    list_pos p = list_first(l);
    for (int i = 0; i < 2; i++) {
        int *v = int_create(20 + i);
        p = list_insert(l, v, p);
        p = list_next(l, p);
    }
    list_print(l, print_int);




    // Clean up the list, including payload
    list_kill(l);
    return 0;
}
```



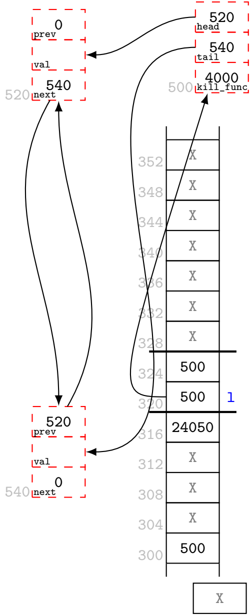
Exempel:

```c
/**
 * list_first() - Return the first position of a list, i.
 *                position of the first element in the li
 * @l: List to inspect.
 *
 * Returns: The first position in the given list.
 */
list_pos list_first(const list *l)
{
    // First position is position of first element.
    return l->head->next;
}
```



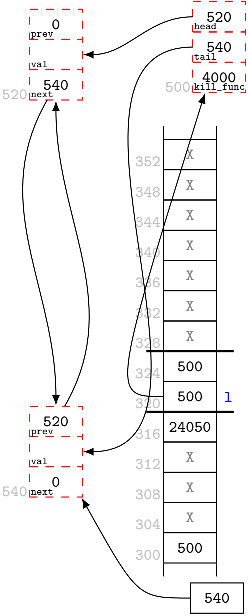
Exempel:

```c
/**
 * list_first() - Return the first position of a list, i.
 *                position of the first element in the li
 * @l: List to inspect.
 *
 * Returns: The first position in the given list.
 */
list_pos list_first(const list *l)
{
    // First position is position of first element.
    return l->head->next;
}
```



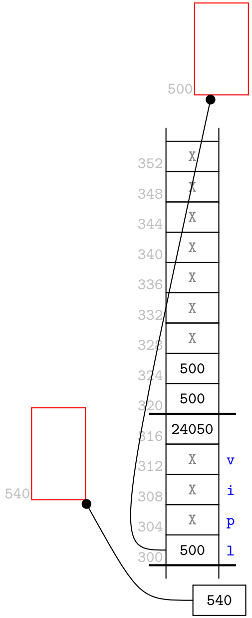
Exempel:

```c
/**
 * list_first() - Return the first position of a list, i.
 *                position of the first element in the li
 * @l: List to inspect.
 *
 * Returns: The first position in the given list.
 */
list_pos list_first(const list *l)
{
    // First position is position of first element.
    return l->head->next;
}
```



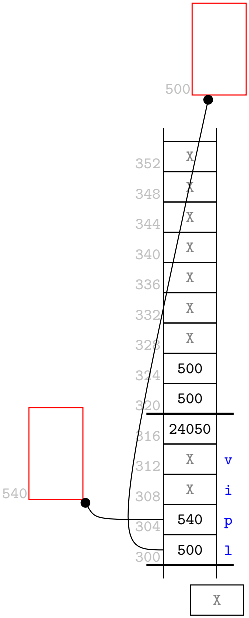
Exempel:

```c
int *int_create(int v)
{
    int *p = malloc(sizeof(*p)); // allocate payload
    *p = v; // assign value
    return p;
}
void int_kill(void *v)
{
    free(v); // deallocate payload
}
int main(void)
{
    // Create empty list, hand over responsibility to
    // deallocate payload using int_kill
    list *l = list_empty(int_kill);
    list_pos p = list_first(l);
    for (int i = 0; i < 2; i++) {
        int *v = int_create(20 + i);
        p = list_insert(l, v, p);
        p = list_next(l, p);
    }
    list_print(l, print_int);






    // Clean up the list, including payload
    list_kill(l);
    return 0;
}
```



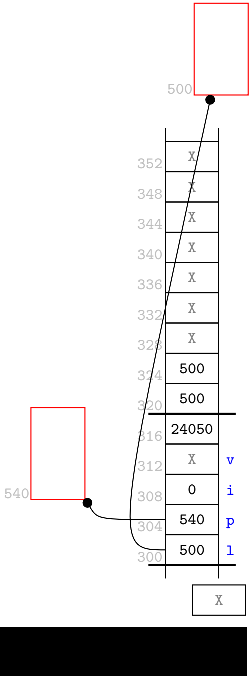
Exempel:

```c
int *int_create(int v)
{
    int *p = malloc(sizeof(*p)); // allocate payload
    *p = v; // assign value
    return p;
}
void int_kill(void *v)
{
    free(v); // deallocate payload
}
int main(void)
{
    // Create empty list, hand over responsibility to
    // deallocate payload using int_kill
    list *l = list_empty(int_kill);
    list_pos p = list_first(l);
    for (int i = 0; i < 2; i++) {
        int *v = int_create(20 + i);
        p = list_insert(l, v, p);
        p = list_next(l, p);
    }
    list_print(l, print_int);





    // Clean up the list, including payload
    list_kill(l);
    return 0;
}
```



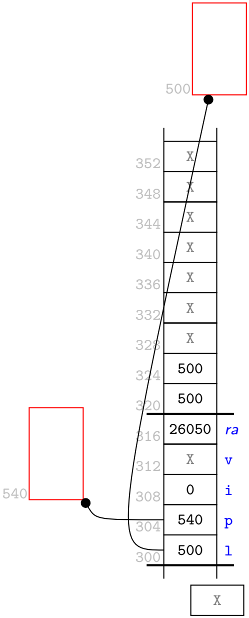
Exempel:

```c
int *int_create(int v)
{
    int *p = malloc(sizeof(*p)); // allocate payload
    *p = v; // assign value
    return p;
}
void int_kill(void *v)
{
    free(v); // deallocate payload
}
int main(void)
{
    // Create empty list, hand over responsibility to
    // deallocate payload using int_kill
    list *l = list_empty(int_kill);
    list_pos p = list_first(l);
    for (int i = 0; i < 2; i++) {
        int *v = int_create(20 + i);
        p = list_insert(l, v, p);
        p = list_next(l, p);
    }
    list_print(l, print_int);





    // Clean up the list, including payload
    list_kill(l);
    return 0;
}
```



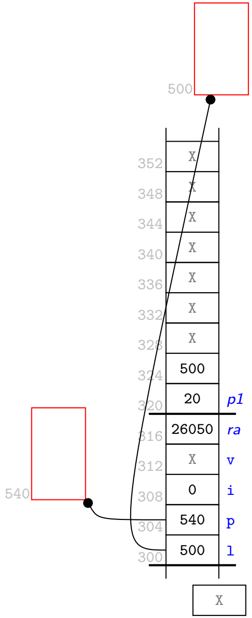
Exempel:

```c
int *int_create(int v)
{
    int *p = malloc(sizeof(*p)); // allocate payload
    *p = v; // assign value
    return p;
}
void int_kill(void *v)
{
    free(v); // deallocate payload
}
int main(void)
{
    // Create empty list, hand over responsibility to
    // deallocate payload using int_kill
    list *l = list_empty(int_kill);
    list_pos p = list_first(l);
    for (int i = 0; i < 2; i++) {
        int *v = int_create(20 + i);
        p = list_insert(l, v, p);
        p = list_next(l, p);
    }
    list_print(l, print_int);




    // Clean up the list, including payload
    list_kill(l);
    return 0;
}
```



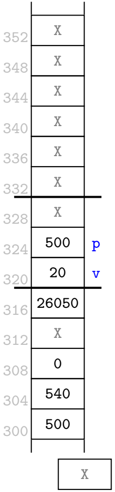
Exempel:

```c
int *int_create(int v)
{
    int *p = malloc(sizeof(*p)); // allocate payload
    *p = v; // assign value
    return p;
}
void int_kill(void *v)
{
    free(v); // deallocate payload
}
int main(void)
{
    // Create empty list, hand over responsibility to
    // deallocate payload using int_kill
    list *l = list_empty(int_kill);
    list_pos p = list_first(l);
    for (int i = 0; i < 2; i++) {
        int *v = int_create(20 + i);
        p = list_insert(l, v, p);
        p = list_next(l, p);
    }
    list_print(l, print_int);





    // Clean up the list, including payload
    list_kill(l);
    return 0;
}
```



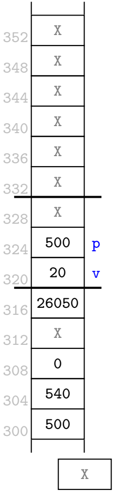
Exempel:

```c
int *int_create(int v)
{
    int *p = malloc(sizeof(*p)); // allocate payload
    *p = v; // assign value
    return p;
}
void int_kill(void *v)
{
    free(v); // deallocate payload
}
int main(void)
{
    // Create empty list, hand over responsibility to
    // deallocate payload using int_kill
    list *l = list_empty(int_kill);
    list_pos p = list_first(l);
    for (int i = 0; i < 2; i++) {
        int *v = int_create(20 + i);
        p = list_insert(l, v, p);
        p = list_next(l, p);
    }
    list_print(l, print_int);




    // Clean up the list, including payload
    list_kill(l);
    return 0;
}
```



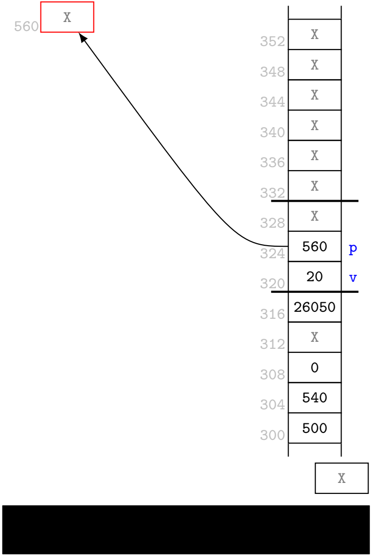
Exempel:

```
int *int_create(int v)
{
    int *p = malloc(sizeof(*p)); // allocate payload
    *p = v; // assign value
    return p;
}
void int_kill(void *v)
{
    free(v); // deallocate payload
}
int main(void)
{
    // Create empty list, hand over responsibility to
    // deallocate payload using int_kill
    list *l = list_empty(int_kill);
    list_pos p = list_first(l);
    for (int i = 0; i < 2; i++) {
        int *v = int_create(20 + i);
        p = list_insert(l, v, p);
        p = list_next(l, p);
    }
    list_print(l, print_int);






    // Clean up the list, including payload
    list_kill(l);
    return 0;
}
```



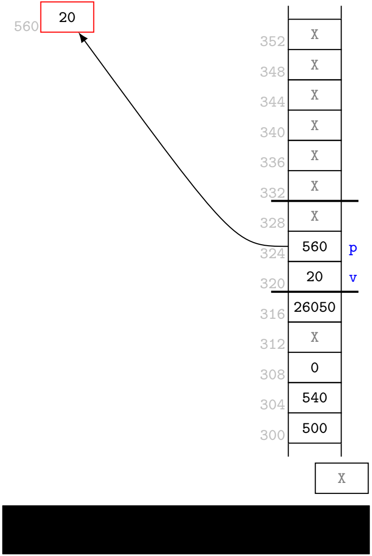
Exempel:

```c
int *int_create(int v)
{
    int *p = malloc(sizeof(*p)); // allocate payload
    *p = v; // assign value
    return p;
}
void int_kill(void *v)
{
    free(v); // deallocate payload
}
int main(void)
{
    // Create empty list, hand over responsibility to
    // deallocate payload using int_kill
    list *l = list_empty(int_kill);
    list_pos p = list_first(l);
    for (int i = 0; i < 2; i++) {
        int *v = int_create(20 + i);
        p = list_insert(l, v, p);
        p = list_next(l, p);
    }
    list_print(l, print_int);






    // Clean up the list, including payload
    list_kill(l);
    return 0;
}
```



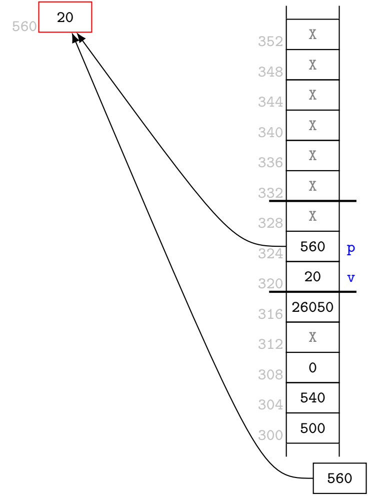
Exempel:

```c
int *int_create(int v)
{
    int *p = malloc(sizeof(*p)); // allocate payload
    *p = v; // assign value
    return p;
}
void int_kill(void *v)
{
    free(v); // deallocate payload
}
int main(void)
{
    // Create empty list, hand over responsibility to
    // deallocate payload using int_kill
    list *l = list_empty(int_kill);
    list_pos p = list_first(l);
    for (int i = 0; i < 2; i++) {
        int *v = int_create(20 + i);
        p = list_insert(l, v, p);
        p = list_next(l, p);
    }
    list_print(l, print_int);






    // Clean up the list, including payload
    list_kill(l);
    return 0;
}
```



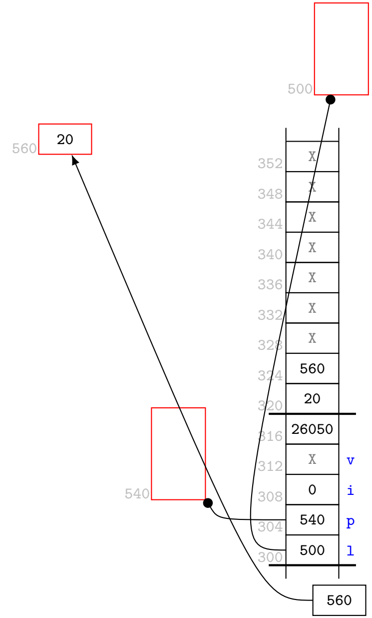
Exempel:

```c
int *int_create(int v)
{
    int *p = malloc(sizeof(*p)); // allocate payload
    *p = v; // assign value
    return p;
}
void int_kill(void *v)
{
    free(v); // deallocate payload
}
int main(void)
{
    // Create empty list, hand over responsibility to
    // deallocate payload using int_kill
    list *l = list_empty(int_kill);
    list_pos p = list_first(l);
    for (int i = 0; i < 2; i++) {
        int *v = int_create(20 + i);
        p = list_insert(l, v, p);
        p = list_next(l, p);
    }
    list_print(l, print_int);





    // Clean up the list, including payload
    list_kill(l);
    return 0;
}
```



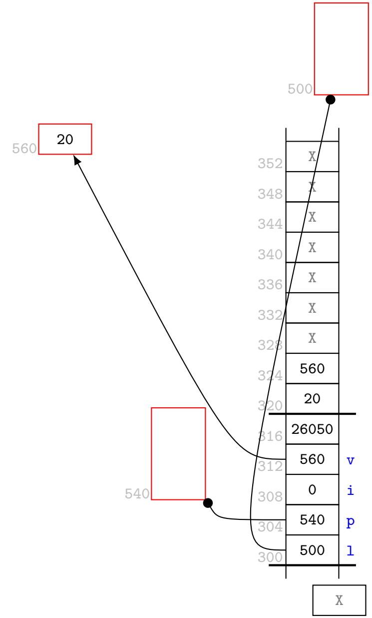
Exempel:

```c
int *int_create(int v)
{
    int *p = malloc(sizeof(*p)); // allocate payload
    *p = v; // assign value
    return p;
}
void int_kill(void *v)
{
    free(v); // deallocate payload
}
int main(void)
{
    // Create empty list, hand over responsibility to
    // deallocate payload using int_kill
    list *l = list_empty(int_kill);
    list_pos p = list_first(l);
    for (int i = 0; i < 2; i++) {
        int *v = int_create(20 + i);
        p = list_insert(l, v, p);
        p = list_next(l, p);
    }
    list_print(l, print_int);







    // Clean up the list, including payload
    list_kill(l);
    return 0;
}
```



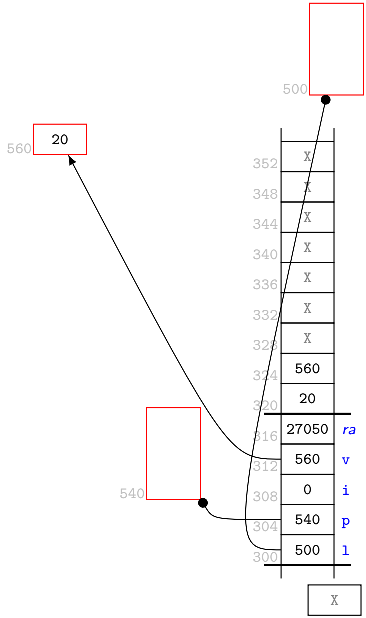
Exempel:

```c
int *int_create(int v)
{
    int *p = malloc(sizeof(*p)); // allocate payload
    *p = v; // assign value
    return p;
}
void int_kill(void *v)
{
    free(v); // deallocate payload
}
int main(void)
{
    // Create empty list, hand over responsibility to
    // deallocate payload using int_kill
    list *l = list_empty(int_kill);
    list_pos p = list_first(l);
    for (int i = 0; i < 2; i++) {
        int *v = int_create(20 + i);
        p = list_insert(l, v, p);
        p = list_next(l, p);
    }
    list_print(l, print_int);






    // Clean up the list, including payload
    list_kill(l);
    return 0;
}
```



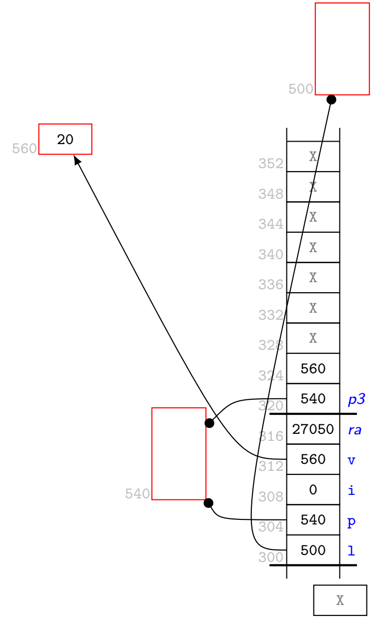
Exempel:

```c
int *int_create(int v)
{
    int *p = malloc(sizeof(*p)); // allocate payload
    *p = v; // assign value
    return p;
}
void int_kill(void *v)
{
    free(v); // deallocate payload
}
int main(void)
{
    // Create empty list, hand over responsibility to
    // deallocate payload using int_kill
    list *l = list_empty(int_kill);
    list_pos p = list_first(l);
    for (int i = 0; i < 2; i++) {
        int *v = int_create(20 + i);
        p = list_insert(l, v, p);
        p = list_next(l, p);
    }
    list_print(l, print_int);






    // Clean up the list, including payload
    list_kill(l);
    return 0;
}
```



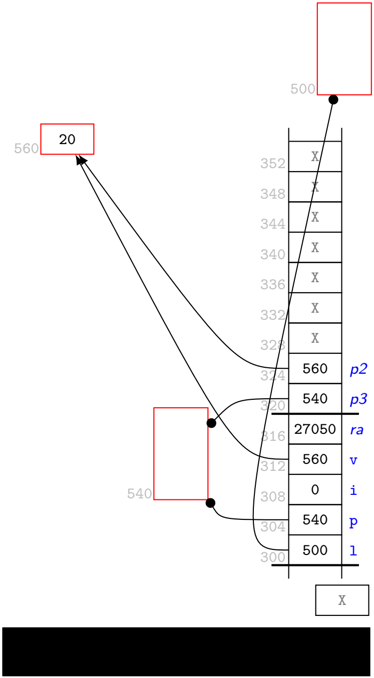
Exempel:

```c
int *int_create(int v)
{
    int *p = malloc(sizeof(*p)); // allocate payload
    *p = v; // assign value
    return p;
}
void int_kill(void *v)
{
    free(v); // deallocate payload
}
int main(void)
{
    // Create empty list, hand over responsibility to
    // deallocate payload using int_kill
    list *l = list_empty(int_kill);
    list_pos p = list_first(l);
    for (int i = 0; i < 2; i++) {
        int *v = int_create(20 + i);
        p = list_insert(l, v, p);
        p = list_next(l, p);
    }
    list_print(l, print_int);
    // Clean up the list, including payload
    list_kill(l);
    return 0;
}
```



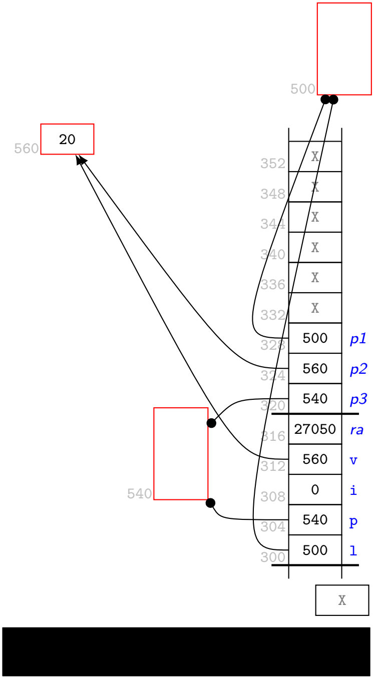
Exempel:

```c
int *int_create(int v)
{
    int *p = malloc(sizeof(*p)); // allocate payload
    *p = v; // assign value
    return p;
}
void int_kill(void *v)
{
    free(v); // deallocate payload
}
int main(void)
{
    // Create empty list, hand over responsibility to
    // deallocate payload using int_kill
    list *l = list_empty(int_kill);
    list_pos p = list_first(l);
    for (int i = 0; i < 2; i++) {
        int *v = int_create(20 + i);
        p = list_insert(l, v, p);
        p = list_next(l, p);
    }
    list_print(l, print_int);





    // Clean up the list, including payload
    list_kill(l);
    return 0;
}
```



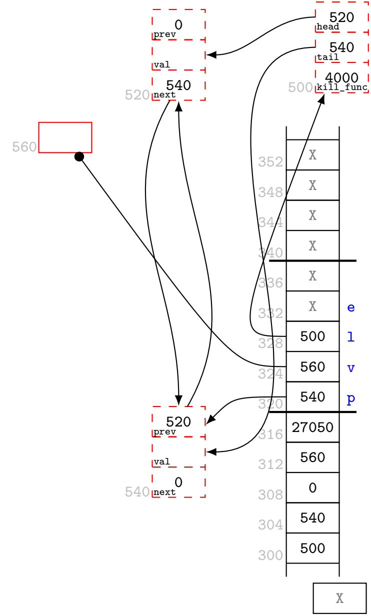
Exempel:

```c
int *int_create(int v)
{
    int *p = malloc(sizeof(*p)); // allocate payload
    *p = v; // assign value
    return p;
}
void int_kill(void *v)
{
    free(v); // deallocate payload
}
int main(void)
{
    // Create empty list, hand over responsibility to
    // deallocate payload using int_kill
    list *l = list_empty(int_kill);
    list_pos p = list_first(l);
    for (int i = 0; i < 2; i++) {
        int *v = int_create(20 + i);
        p = list_insert(l, v, p);
        p = list_next(l, p);
    }
    list_print(l, print_int);




    // Clean up the list, including payload
    list_kill(l);
    return 0;
}
```



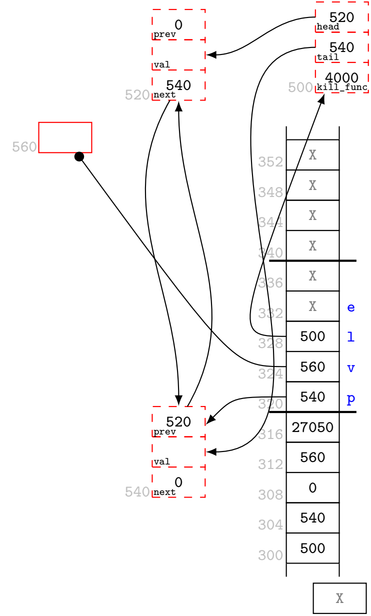
Exempel:

```c
/**
 * list_insert() - Insert a new element with a given valu
 * @l: List to manipulate.
 * @v: Value (pointer) to be inserted into the list.
 * @p: Position in the list before which the value should
 *
 * Creates a new element and inserts it into the list bef
 * Stores data in the new element.
 *
 * Returns: The position of the newly created element.
 */
list_pos list_insert(list *l, void *v, const list_pos p)
{
    // Allocate memory for a new cell.
    list_pos e = malloc(sizeof(cell));

    // Store the value.
    e->val = v;
    // Add links to/from the new cell.
    e->next = p;
    e->prev = p->prev;
    e->prev->next = e;
    p->prev = e;

    // Return the position of the new cell.
    return e;
}
```



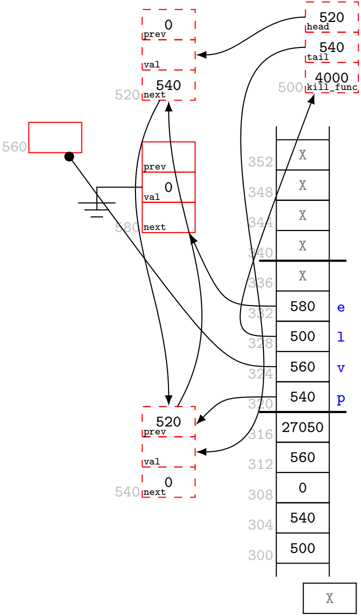
Exempel:

```c
/**
 * list_insert() - Insert a new element with a given valu
 * @l: List to manipulate.
 * @v: Value (pointer) to be inserted into the list.
 * @p: Position in the list before which the value should
 *
 * Creates a new element and inserts it into the list bef
 * Stores data in the new element.
 *
 * Returns: The position of the newly created element.
 */
list_pos list_insert(list *l, void *v, const list_pos p)
{
    // Allocate memory for a new cell.
    list_pos e = malloc(sizeof(cell));

    // Store the value.
    e->val = v;
    // Add links to/from the new cell.
    e->next = p;
    e->prev = p->prev;
    e->prev->next = e;
    p->prev = e;

    // Return the position of the new cell.
    return e;
}
```



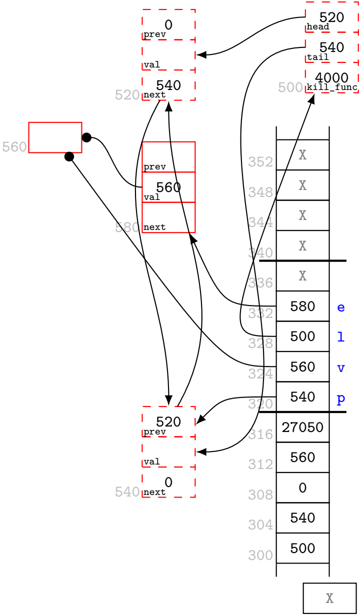
Exempel:

```c
/**
 * list_insert() - Insert a new element with a given valu
 * @l: List to manipulate.
 * @v: Value (pointer) to be inserted into the list.
 * @p: Position in the list before which the value should
 *
 * Creates a new element and inserts it into the list bef
 * Stores data in the new element.
 *
 * Returns: The position of the newly created element.
 */
list_pos list_insert(list *l, void *v, const list_pos p)
{
    // Allocate memory for a new cell.
    list_pos e = malloc(sizeof(cell));

    // Store the value.
    e->val = v;
    // Add links to/from the new cell.
    e->next = p;
    e->prev = p->prev;
    e->prev->next = e;
    p->prev = e;

    // Return the position of the new cell.
    return e;
}
```



Exempel:

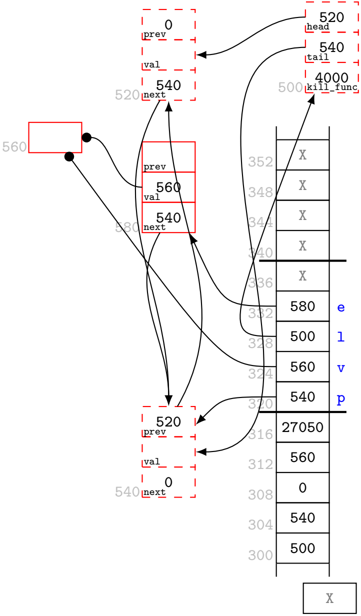
```c
/**
 * list_insert() - Insert a new element with a given valu
 * @l: List to manipulate.
 * @v: Value (pointer) to be inserted into the list.
 * @p: Position in the list before which the value should
 *
 * Creates a new element and inserts it into the list bef
 * Stores data in the new element.
 *
 * Returns: The position of the newly created element.
 */
list_pos list_insert(list *l, void *v, const list_pos p)
{
    // Allocate memory for a new cell.
    list_pos e = malloc(sizeof(cell));

    // Store the value.
    e->val = v;
    // Add links to/from the new cell.
    e->next = p;
    e->prev = p->prev;
    e->prev->next = e;
    p->prev = e;

    // Return the position of the new cell.
    return e;
}
```



[redacted]

Exempel:

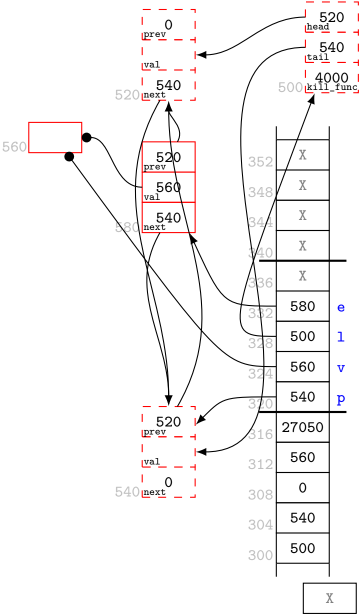
```c
/**
 * list_insert() - Insert a new element with a given valu
 * @l: List to manipulate.
 * @v: Value (pointer) to be inserted into the list.
 * @p: Position in the list before which the value should
 *
 * Creates a new element and inserts it into the list bef
 * Stores data in the new element.
 *
 * Returns: The position of the newly created element.
 */
list_pos list_insert(list *l, void *v, const list_pos p)
{
    // Allocate memory for a new cell.
    list_pos e = malloc(sizeof(cell));

    // Store the value.
    e->val = v;
    // Add links to/from the new cell.
    e->next = p;
    e->prev = p->prev;
    e->prev->next = e;
    p->prev = e;

    // Return the position of the new cell.
    return e;
}
```



[redacted]

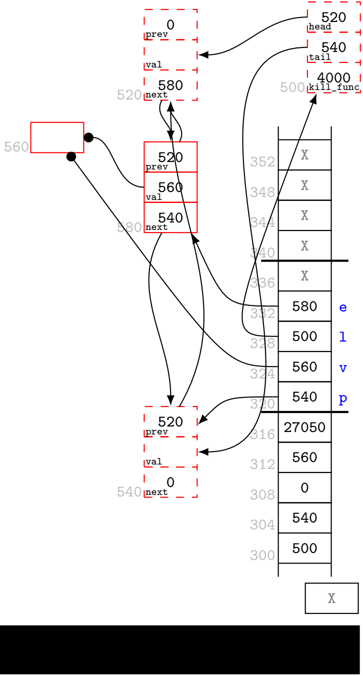
Exempel:

```c
/**
 * list_insert() - Insert a new element with a given valu
 * @l: List to manipulate.
 * @v: Value (pointer) to be inserted into the list.
 * @p: Position in the list before which the value should
 *
 * Creates a new element and inserts it into the list bef
 * Stores data in the new element.
 *
 * Returns: The position of the newly created element.
 */
list_pos list_insert(list *l, void *v, const list_pos p)
{
    // Allocate memory for a new cell.
    list_pos e = malloc(sizeof(cell));

    // Store the value.
    e->val = v;
    // Add links to/from the new cell.
    e->next = p;
    e->prev = p->prev;
    e->prev->next = e;
    p->prev = e;

    // Return the position of the new cell.
    return e;
}
```



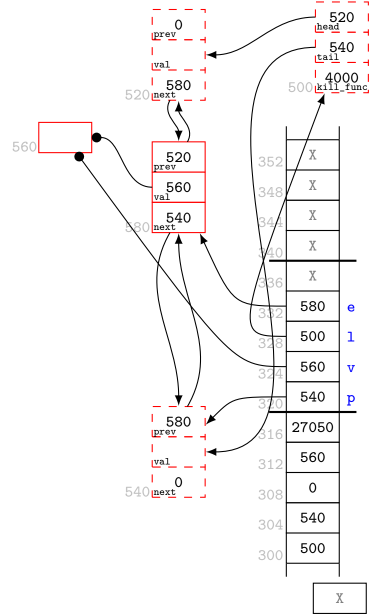
Exempel:

```c
/**
 * list_insert() - Insert a new element with a given valu
 * @l: List to manipulate.
 * @v: Value (pointer) to be inserted into the list.
 * @p: Position in the list before which the value should
 *
 * Creates a new element and inserts it into the list bef
 * Stores data in the new element.
 *
 * Returns: The position of the newly created element.
 */
list_pos list_insert(list *l, void *v, const list_pos p)
{
    // Allocate memory for a new cell.
    list_pos e = malloc(sizeof(cell));

    // Store the value.
    e->val = v;
    // Add links to/from the new cell.
    e->next = p;
    e->prev = p->prev;
    e->prev->next = e;
    p->prev = e;

    // Return the position of the new cell.
    return e;
}
```



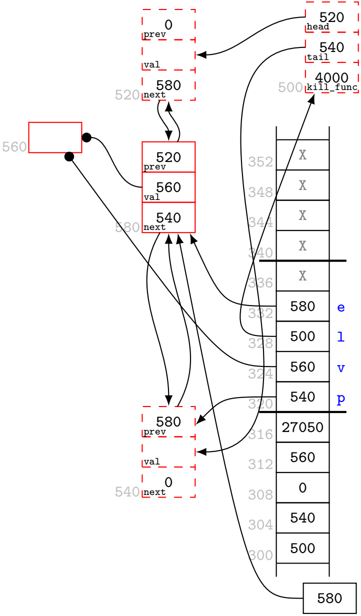
Exempel:

```c
/**
 * list_insert() - Insert a new element with a given valu
 * @l: List to manipulate.
 * @v: Value (pointer) to be inserted into the list.
 * @p: Position in the list before which the value should
 *
 * Creates a new element and inserts it into the list bef
 * Stores data in the new element.
 *
 * Returns: The position of the newly created element.
 */
list_pos list_insert(list *l, void *v, const list_pos p)
{
    // Allocate memory for a new cell.
    list_pos e = malloc(sizeof(cell));

    // Store the value.
    e->val = v;
    // Add links to/from the new cell.
    e->next = p;
    e->prev = p->prev;
    e->prev->next = e;
    p->prev = e;

    // Return the position of the new cell.
    return e;
}
```



Exempel:

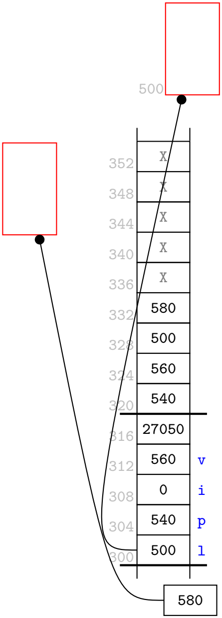
```c
/**
 * list_insert() - Insert a new element with a given valu
 * @l: List to manipulate.
 * @v: Value (pointer) to be inserted into the list.
 * @p: Position in the list before which the value should
 *
 * Creates a new element and inserts it into the list bef
 * Stores data in the new element.
 *
 * Returns: The position of the newly created element.
 */
list_pos list_insert(list *l, void *v, const list_pos p)
{
    // Allocate memory for a new cell.
    list_pos e = malloc(sizeof(cell));

    // Store the value.
    e->val = v;
    // Add links to/from the new cell.
    e->next = p;
    e->prev = p->prev;
    e->prev->next = e;
    p->prev = e;

    // Return the position of the new cell.
    return e;
}
```



[redacted]

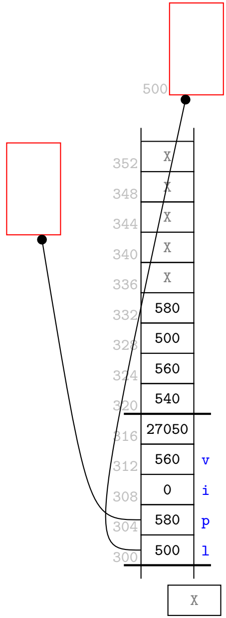
Exempel:

```c
int *int_create(int v)
{
    int *p = malloc(sizeof(*p)); // allocate payload
    *p = v; // assign value
    return p;
}
void int_kill(void *v)
{
    free(v); // deallocate payload
}
int main(void)
{
    // Create empty list, hand over responsibility to
    // deallocate payload using int_kill
    list *l = list_empty(int_kill);
    list_pos p = list_first(l);
    for (int i = 0; i < 2; i++) {
        int *v = int_create(20 + i);
        p = list_insert(l, v, p);
        p = list_next(l, p);
    }
    list_print(l, print_int);

    // Clean up the list, including payload
    list_kill(l);
    return 0;
}
```



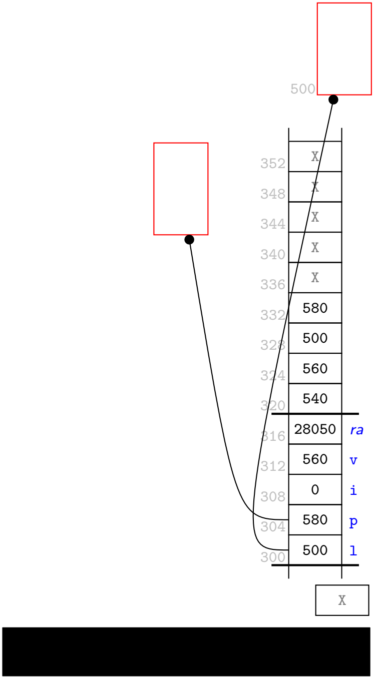
Exempel:

```c
int *int_create(int v)
{
    int *p = malloc(sizeof(*p)); // allocate payload
    *p = v; // assign value
    return p;
}
void int_kill(void *v)
{
    free(v); // deallocate payload
}
int main(void)
{
    // Create empty list, hand over responsibility to
    // deallocate payload using int_kill
    list *l = list_empty(int_kill);
    list_pos p = list_first(l);
    for (int i = 0; i < 2; i++) {
        int *v = int_create(20 + i);
        p = list_insert(l, v, p);
        p = list_next(l, p);
    }
    list_print(l, print_int);






    // Clean up the list, including payload
    list_kill(l);
    return 0;
}
```



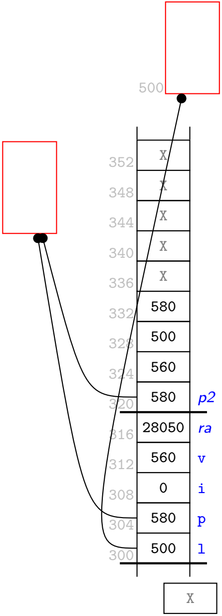
Exempel:

```c
int *int_create(int v)
{
    int *p = malloc(sizeof(*p)); // allocate payload
    *p = v; // assign value
    return p;
}
void int_kill(void *v)
{
    free(v); // deallocate payload
}
int main(void)
{
    // Create empty list, hand over responsibility to
    // deallocate payload using int_kill
    list *l = list_empty(int_kill);
    list_pos p = list_first(l);
    for (int i = 0; i < 2; i++) {
        int *v = int_create(20 + i);
        p = list_insert(l, v, p);
        p = list_next(l, p);
    }
    list_print(l, print_int);






    // Clean up the list, including payload
    list_kill(l);
    return 0;
}
```



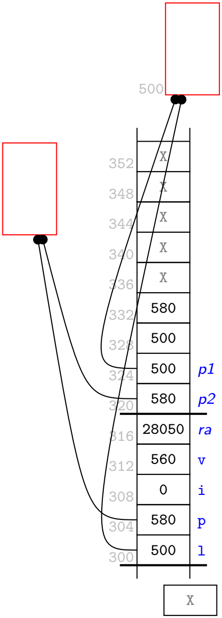
Exempel:

```c
int *int_create(int v)
{
    int *p = malloc(sizeof(*p)); // allocate payload
    *p = v; // assign value
    return p;
}
void int_kill(void *v)
{
    free(v); // deallocate payload
}
int main(void)
{
    // Create empty list, hand over responsibility to
    // deallocate payload using int_kill
    list *l = list_empty(int_kill);
    list_pos p = list_first(l);
    for (int i = 0; i < 2; i++) {
        int *v = int_create(20 + i);
        p = list_insert(l, v, p);
        p = list_next(l, p);
    }
    list_print(l, print_int);





    // Clean up the list, including payload
    list_kill(l);
    return 0;
}
```



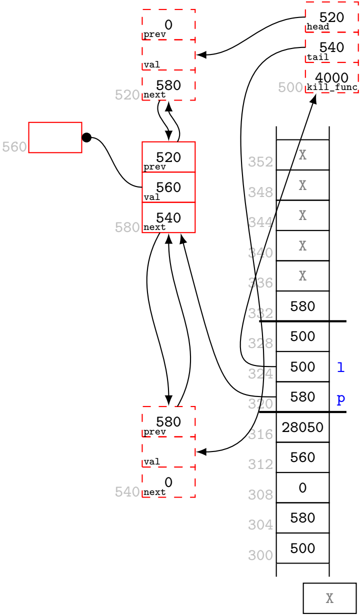
Exempel:

```c
int *int_create(int v)
{
    int *p = malloc(sizeof(*p)); // allocate payload
    *p = v; // assign value
    return p;
}
void int_kill(void *v)
{
    free(v); // deallocate payload
}
int main(void)
{
    // Create empty list, hand over responsibility to
    // deallocate payload using int_kill
    list *l = list_empty(int_kill);
    list_pos p = list_first(l);
    for (int i = 0; i < 2; i++) {
        int *v = int_create(20 + i);
        p = list_insert(l, v, p);
        p = list_next(l, p);
    }
    list_print(l, print_int);






    // Clean up the list, including payload
    list_kill(l);
    return 0;
}
```



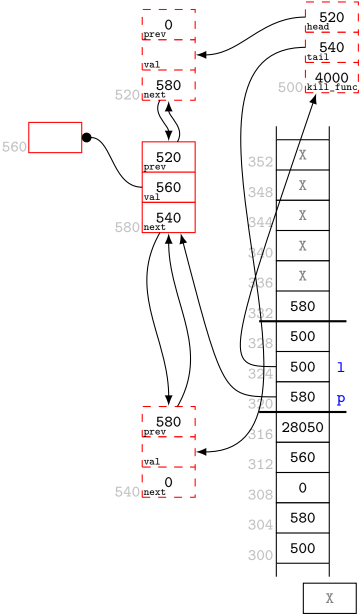
Exempel:

```c
/**
 * list_next() - Return the next position in a list.
 * @l: List to inspect.
 * @p: Any valid position except the last in the list.
 *
 * Returns: The position in the list after the given posi
 *          NOTE: The return value is undefined for the l
 */
list_pos list_next(const list *l, const list_pos p)
{
    return p->next;
}
```



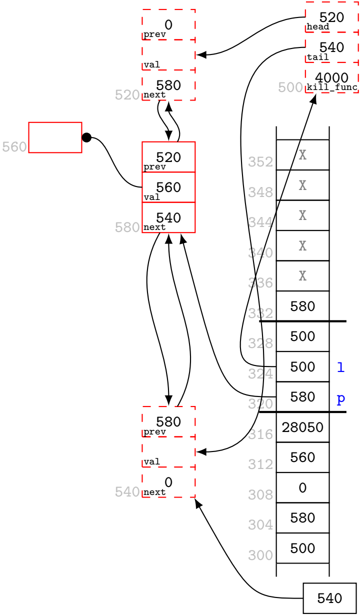
Exempel:

```c
/**
 * list_next() - Return the next position in a list.
 * @l: List to inspect.
 * @p: Any valid position except the last in the list.
 *
 * Returns: The position in the list after the given posi
 *          NOTE: The return value is undefined for the l
 */
list_pos list_next(const list *l, const list_pos p)
{
    return p->next;
}
```



Exempel:

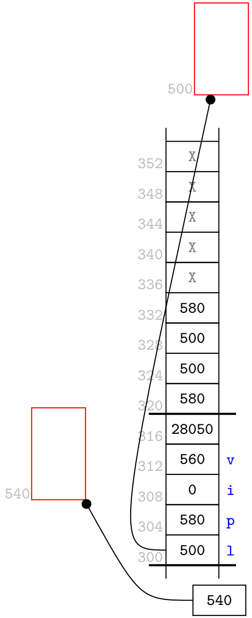
```c
/**
 * list_next() - Return the next position in a list.
 * @l: List to inspect.
 * @p: Any valid position except the last in the list.
 *
 * Returns: The position in the list after the given posi
 *          NOTE: The return value is undefined for the l
 */
list_pos list_next(const list *l, const list_pos p)
{
    return p->next;
}
```



[redacted]

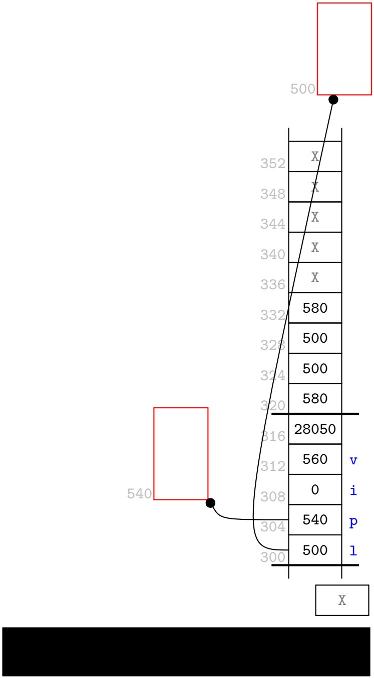
Exempel:

```c
int *int_create(int v)
{
    int *p = malloc(sizeof(*p)); // allocate payload
    *p = v; // assign value
    return p;
}
void int_kill(void *v)
{
    free(v); // deallocate payload
}
int main(void)
{
    // Create empty list, hand over responsibility to
    // deallocate payload using int_kill
    list *l = list_empty(int_kill);
    list_pos p = list_first(l);
    for (int i = 0; i < 2; i++) {
        int *v = int_create(20 + i);
        p = list_insert(l, v, p);
        p = list_next(l, p);
    }
    list_print(l, print_int);





    // Clean up the list, including payload
    list_kill(l);
    return 0;
}
```



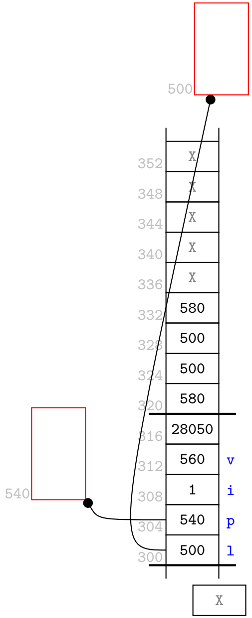
Exempel:

```c
int *int_create(int v)
{
    int *p = malloc(sizeof(*p)); // allocate payload
    *p = v; // assign value
    return p;
}
void int_kill(void *v)
{
    free(v); // deallocate payload
}
int main(void)
{
    // Create empty list, hand over responsibility to
    // deallocate payload using int_kill
    list *l = list_empty(int_kill);
    list_pos p = list_first(l);
    for (int i = 0; i < 2; i++) {
        int *v = int_create(20 + i);
        p = list_insert(l, v, p);
        p = list_next(l, p);
    }
    list_print(l, print_int);





    // Clean up the list, including payload
    list_kill(l);
    return 0;
}
```



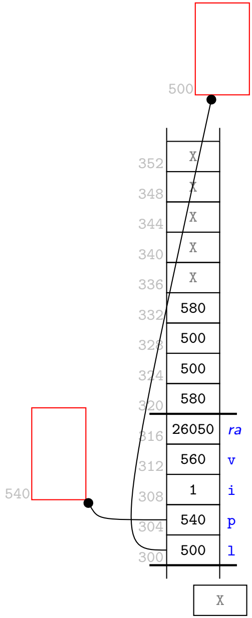
Exempel:

```c
int *int_create(int v)
{
    int *p = malloc(sizeof(*p)); // allocate payload
    *p = v; // assign value
    return p;
}
void int_kill(void *v)
{
    free(v); // deallocate payload
}
int main(void)
{
    // Create empty list, hand over responsibility to
    // deallocate payload using int_kill
    list *l = list_empty(int_kill);
    list_pos p = list_first(l);
    for (int i = 0; i < 2; i++) {
        int *v = int_create(20 + i);
        p = list_insert(l, v, p);
        p = list_next(l, p);
    }
    list_print(l, print_int);




    
    // Clean up the list, including payload
    list_kill(l);
    return 0;
}
```



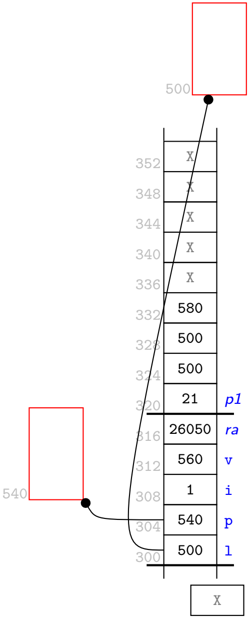
Exempel:

```c
int *int_create(int v)
{
    int *p = malloc(sizeof(*p)); // allocate payload
    *p = v; // assign value
    return p;
}
void int_kill(void *v)
{
    free(v); // deallocate payload
}
int main(void)
{
    // Create empty list, hand over responsibility to
    // deallocate payload using int_kill
    list *l = list_empty(int_kill);
    list_pos p = list_first(l);
    for (int i = 0; i < 2; i++) {
        int *v = int_create(20 + i);
        p = list_insert(l, v, p);
        p = list_next(l, p);
    }
    list_print(l, print_int);




    // Clean up the list, including payload
    list_kill(l);
    return 0;
}
```



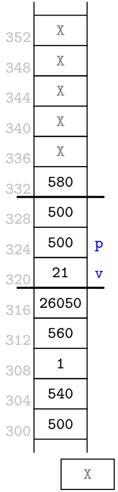
Exempel:

```c
int *int_create(int v)
{
    int *p = malloc(sizeof(*p)); // allocate payload
    *p = v; // assign value
    return p;
}
void int_kill(void *v)
{
    free(v); // deallocate payload
}
int main(void)
{
    // Create empty list, hand over responsibility to
    // deallocate payload using int_kill
    list *l = list_empty(int_kill);
    list_pos p = list_first(l);
    for (int i = 0; i < 2; i++) {
        int *v = int_create(20 + i);
        p = list_insert(l, v, p);
        p = list_next(l, p);
    }
    list_print(l, print_int);






    // Clean up the list, including payload
    list_kill(l);
    return 0;
}
```



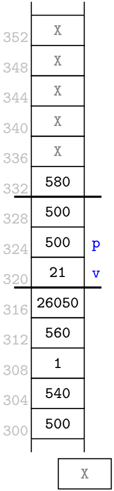
Exempel:

```c
int *int_create(int v)
{
    int *p = malloc(sizeof(*p)); // allocate payload
    *p = v; // assign value
    return p;
}
void int_kill(void *v)
{
    free(v); // deallocate payload
}
int main(void)
{
    // Create empty list, hand over responsibility to
    // deallocate payload using int_kill
    list *l = list_empty(int_kill);
    list_pos p = list_first(l);
    for (int i = 0; i < 2; i++) {
        int *v = int_create(20 + i);
        p = list_insert(l, v, p);
        p = list_next(l, p);
    }
    list_print(l, print_int);





    // Clean up the list, including payload
    list_kill(l);
    return 0;
}
```



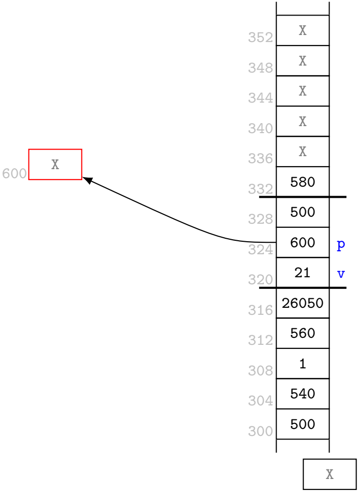
Exempel:

```
int *int_create(int v)
{
    int *p = malloc(sizeof(*p)); // allocate payload
    *p = v; // assign value
    return p;
}
void int_kill(void *v)
{
    free(v); // deallocate payload
}
int main(void)
{
    // Create empty list, hand over responsibility to
    // deallocate payload using int_kill
    list *l = list_empty(int_kill);
    list_pos p = list_first(l);
    for (int i = 0; i < 2; i++) {
        int *v = int_create(20 + i);
        p = list_insert(l, v, p);
        p = list_next(l, p);
    }
    list_print(l, print_int);






    // Clean up the list, including payload
    list_kill(l);
    return 0;
}
```



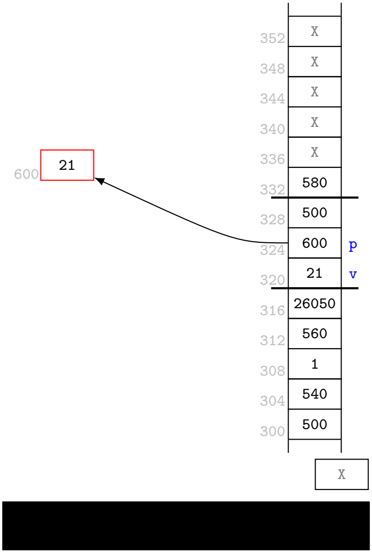
Exempel:

```c
int *int_create(int v)
{
    int *p = malloc(sizeof(*p)); // allocate payload
    *p = v; // assign value
    return p;
}
void int_kill(void *v)
{
    free(v); // deallocate payload
}
int main(void)
{
    // Create empty list, hand over responsibility to
    // deallocate payload using int_kill
    list *l = list_empty(int_kill);
    list_pos p = list_first(l);
    for (int i = 0; i < 2; i++) {
        int *v = int_create(20 + i);
        p = list_insert(l, v, p);
        p = list_next(l, p);
    }
    list_print(l, print_int);





    // Clean up the list, including payload
    list_kill(l);
    return 0;
}
```



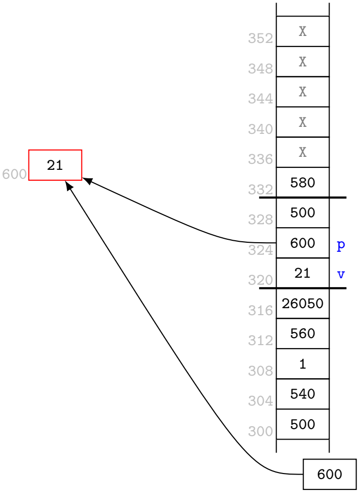
Exempel:

```c
int *int_create(int v)
{
    int *p = malloc(sizeof(*p)); // allocate payload
    *p = v; // assign value
    return p;
}
void int_kill(void *v)
{
    free(v); // deallocate payload
}
int main(void)
{
    // Create empty list, hand over responsibility to
    // deallocate payload using int_kill
    list *l = list_empty(int_kill);
    list_pos p = list_first(l);
    for (int i = 0; i < 2; i++) {
        int *v = int_create(20 + i);
        p = list_insert(l, v, p);
        p = list_next(l, p);
    }
    list_print(l, print_int);

    // Clean up the list, including payload
    list_kill(l);
    return 0;
}
```



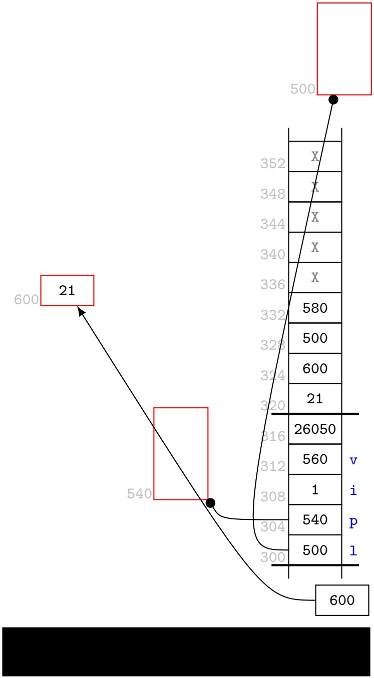
Exempel:

```
int *int_create(int v)
{
    int *p = malloc(sizeof(*p)); // allocate payload
    *p = v; // assign value
    return p;
}
void int_kill(void *v)
{
    free(v); // deallocate payload
}
int main(void)
{
    // Create empty list, hand over responsibility to
    // deallocate payload using int_kill
    list *l = list_empty(int_kill);
    list_pos p = list_first(l);
    for (int i = 0; i < 2; i++) {
        int *v = int_create(20 + i);
        p = list_insert(l, v, p);
        p = list_next(l, p);
    }
    list_print(l, print_int);






    // Clean up the list, including payload
    list_kill(l);
    return 0;
}
```



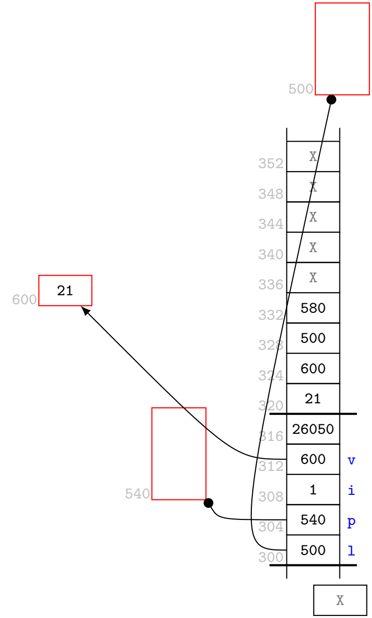
Exempel:

```c
int *int_create(int v)
{
    int *p = malloc(sizeof(*p)); // allocate payload
    *p = v; // assign value
    return p;
}
void int_kill(void *v)
{
    free(v); // deallocate payload
}
int main(void)
{
    // Create empty list, hand over responsibility to
    // deallocate payload using int_kill
    list *l = list_empty(int_kill);
    list_pos p = list_first(l);
    for (int i = 0; i < 2; i++) {
        int *v = int_create(20 + i);
        p = list_insert(l, v, p);
        p = list_next(l, p);
    }
    list_print(l, print_int);




    // Clean up the list, including payload
    list_kill(l);
    return 0;
}
```



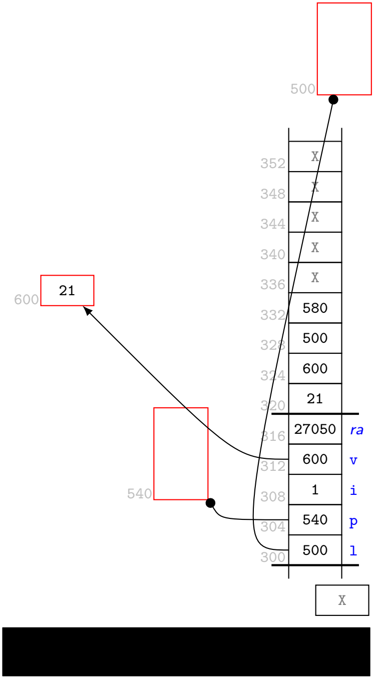
Exempel:

```c
int *int_create(int v)
{
    int *p = malloc(sizeof(*p)); // allocate payload
    *p = v; // assign value
    return p;
}
void int_kill(void *v)
{
    free(v); // deallocate payload
}
int main(void)
{
    // Create empty list, hand over responsibility to
    // deallocate payload using int_kill
    list *l = list_empty(int_kill);
    list_pos p = list_first(l);
    for (int i = 0; i < 2; i++) {
        int *v = int_create(20 + i);
        p = list_insert(l, v, p);
        p = list_next(l, p);
    }
    list_print(l, print_int);




    // Clean up the list, including payload
    list_kill(l);
    return 0;
}
```



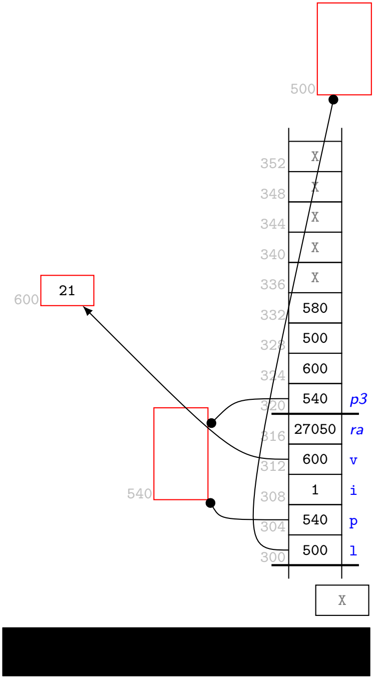
Exempel:

```c
int *int_create(int v)
{
    int *p = malloc(sizeof(*p)); // allocate payload
    *p = v; // assign value
    return p;
}
void int_kill(void *v)
{
    free(v); // deallocate payload
}
int main(void)
{
    // Create empty list, hand over responsibility to
    // deallocate payload using int_kill
    list *l = list_empty(int_kill);
    list_pos p = list_first(l);
    for (int i = 0; i < 2; i++) {
        int *v = int_create(20 + i);
        p = list_insert(l, v, p);
        p = list_next(l, p);
    }
    list_print(l, print_int);





    // Clean up the list, including payload
    list_kill(l);
    return 0;
}
```



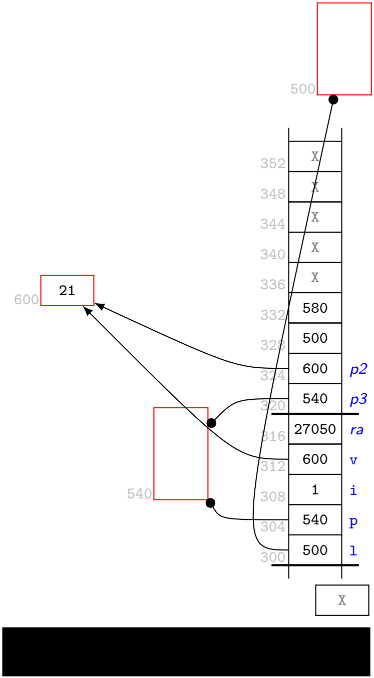
Exempel:

```
int *int_create(int v)
{
    int *p = malloc(sizeof(*p)); // allocate payload
    *p = v; // assign value
    return p;
}
void int_kill(void *v)
{
    free(v); // deallocate payload
}
int main(void)
{
    // Create empty list, hand over responsibility to
    // deallocate payload using int_kill
    list *l = list_empty(int_kill);
    list_pos p = list_first(l);
    for (int i = 0; i < 2; i++) {
        int *v = int_create(20 + i);
        p = list_insert(l, v, p);
        p = list_next(l, p);
    }
    list_print(l, print_int);






    // Clean up the list, including payload
    list_kill(l);
    return 0;
}
```



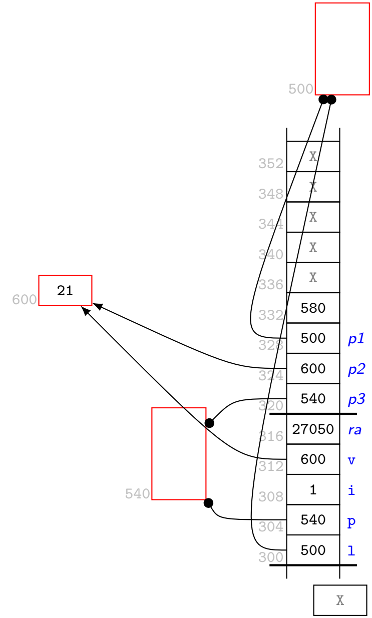
Exempel:

```c
int *int_create(int v)
{
    int *p = malloc(sizeof(*p)); // allocate payload
    *p = v; // assign value
    return p;
}
void int_kill(void *v)
{
    free(v); // deallocate payload
}
int main(void)
{
    // Create empty list, hand over responsibility to
    // deallocate payload using int_kill
    list *l = list_empty(int_kill);
    list_pos p = list_first(l);
    for (int i = 0; i < 2; i++) {
        int *v = int_create(20 + i);
        p = list_insert(l, v, p);
        p = list_next(l, p);
    }
    list_print(l, print_int);






    // Clean up the list, including payload
    list_kill(l);
    return 0;
}
```



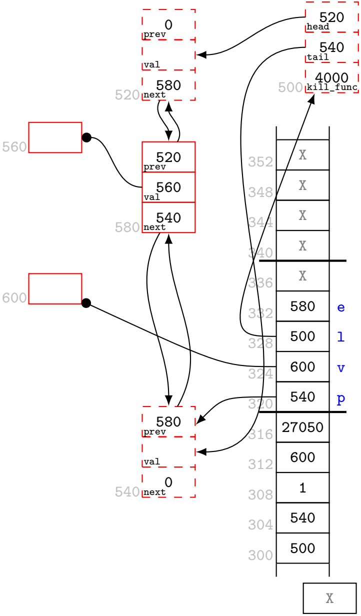
Exempel:

```c
int *int_create(int v)
{
    int *p = malloc(sizeof(*p)); // allocate payload
    *p = v; // assign value
    return p;
}
void int_kill(void *v)
{
    free(v); // deallocate payload
}
int main(void)
{
    // Create empty list, hand over responsibility to
    // deallocate payload using int_kill
    list *l = list_empty(int_kill);
    list_pos p = list_first(l);
    for (int i = 0; i < 2; i++) {
        int *v = int_create(20 + i);
        p = list_insert(l, v, p);
        p = list_next(l, p);
    }
    list_print(l, print_int);




    // Clean up the list, including payload
    list_kill(l);
    return 0;
}
```



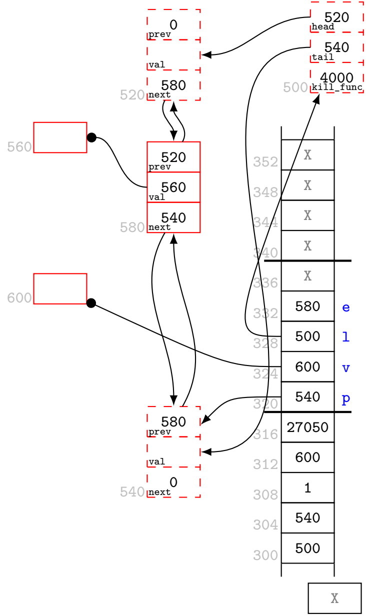
Exempel:

```c
/**
 * list_insert() - Insert a new element with a given valu
 * @l: List to manipulate.
 * @v: Value (pointer) to be inserted into the list.
 * @p: Position in the list before which the value should
 *
 * Creates a new element and inserts it into the list bef
 * Stores data in the new element.
 *
 * Returns: The position of the newly created element.
 */
list_pos list_insert(list *l, void *v, const list_pos p)
{
    // Allocate memory for a new cell.
    list_pos e = malloc(sizeof(cell));

    // Store the value.
    e->val = v;
    // Add links to/from the new cell.
    e->next = p;
    e->prev = p->prev;
    e->prev->next = e;
    p->prev = e;

    // Return the position of the new cell.
    return e;
}
```



Exempel:

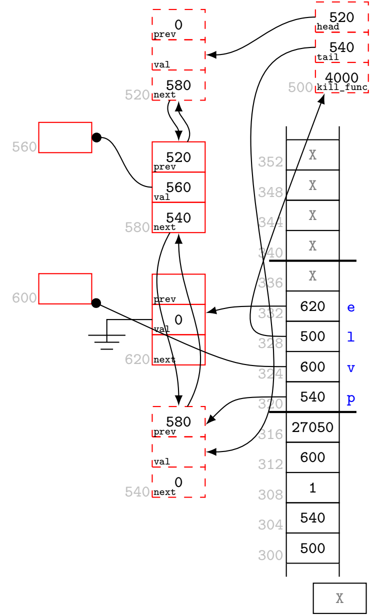
```c
/**
 * list_insert() - Insert a new element with a given valu
 * @l: List to manipulate.
 * @v: Value (pointer) to be inserted into the list.
 * @p: Position in the list before which the value should
 *
 * Creates a new element and inserts it into the list bef
 * Stores data in the new element.
 *
 * Returns: The position of the newly created element.
 */
list_pos list_insert(list *l, void *v, const list_pos p)
{
    // Allocate memory for a new cell.
    list_pos e = malloc(sizeof(cell));

    // Store the value.
    e->val = v;
    // Add links to/from the new cell.
    e->next = p;
    e->prev = p->prev;
    e->prev->next = e;
    p->prev = e;

    // Return the position of the new cell.
    return e;
}
```



[redacted]

Exempel:

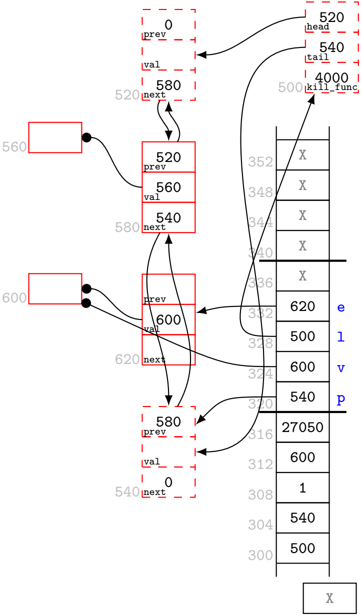
```c
/**
 * list_insert() - Insert a new element with a given valu
 * @l: List to manipulate.
 * @v: Value (pointer) to be inserted into the list.
 * @p: Position in the list before which the value should
 *
 * Creates a new element and inserts it into the list bef
 * Stores data in the new element.
 *
 * Returns: The position of the newly created element.
 */
list_pos list_insert(list *l, void *v, const list_pos p)
{
    // Allocate memory for a new cell.
    list_pos e = malloc(sizeof(cell));

    // Store the value.
    e->val = v;
    // Add links to/from the new cell.
    e->next = p;
    e->prev = p->prev;
    e->prev->next = e;
    p->prev = e;

    // Return the position of the new cell.
    return e;
}
```



[redacted]

Exempel:

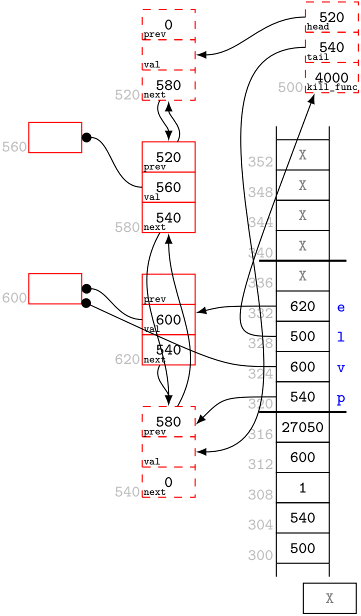
```c
/**
 * list_insert() - Insert a new element with a given valu
 * @l: List to manipulate.
 * @v: Value (pointer) to be inserted into the list.
 * @p: Position in the list before which the value should
 *
 * Creates a new element and inserts it into the list bef
 * Stores data in the new element.
 *
 * Returns: The position of the newly created element.
 */
list_pos list_insert(list *l, void *v, const list_pos p)
{
    // Allocate memory for a new cell.
    list_pos e = malloc(sizeof(cell));

    // Store the value.
    e->val = v;
    // Add links to/from the new cell.
    e->next = p;
    e->prev = p->prev;
    e->prev->next = e;
    p->prev = e;

    // Return the position of the new cell.
    return e;
}
```



[redacted]

Exempel:

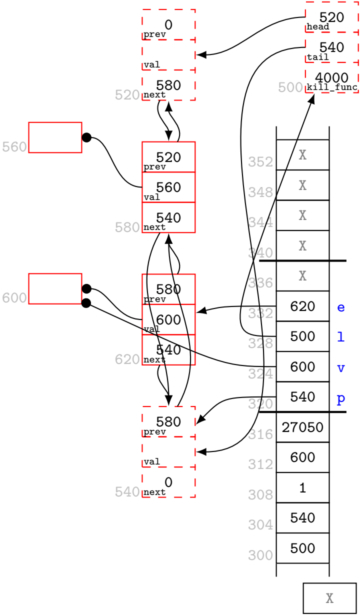
```c
/**
 * list_insert() - Insert a new element with a given valu
 * @l: List to manipulate.
 * @v: Value (pointer) to be inserted into the list.
 * @p: Position in the list before which the value should
 *
 * Creates a new element and inserts it into the list bef
 * Stores data in the new element.
 *
 * Returns: The position of the newly created element.
 */
list_pos list_insert(list *l, void *v, const list_pos p)
{
    // Allocate memory for a new cell.
    list_pos e = malloc(sizeof(cell));

    // Store the value.
    e->val = v;
    // Add links to/from the new cell.
    e->next = p;
    e->prev = p->prev;
    e->prev->next = e;
    p->prev = e;

    // Return the position of the new cell.
    return e;
}
```



[redacted]

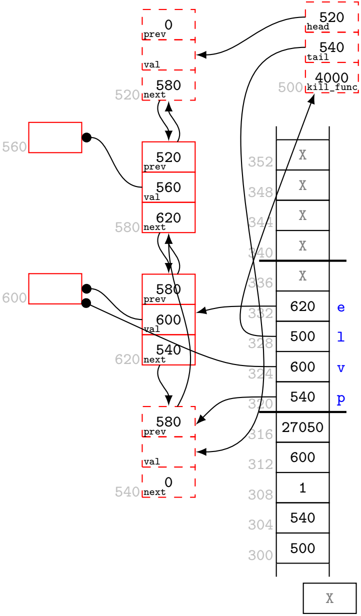
Exempel:

```c
/**
 * list_insert() - Insert a new element with a given valu
 * @l: List to manipulate.
 * @v: Value (pointer) to be inserted into the list.
 * @p: Position in the list before which the value should
 *
 * Creates a new element and inserts it into the list bef
 * Stores data in the new element.
 *
 * Returns: The position of the newly created element.
 */
list_pos list_insert(list *l, void *v, const list_pos p)
{
    // Allocate memory for a new cell.
    list_pos e = malloc(sizeof(cell));

    // Store the value.
    e->val = v;
    // Add links to/from the new cell.
    e->next = p;
    e->prev = p->prev;
    e->prev->next = e;
    p->prev = e;

    // Return the position of the new cell.
    return e;
}
```



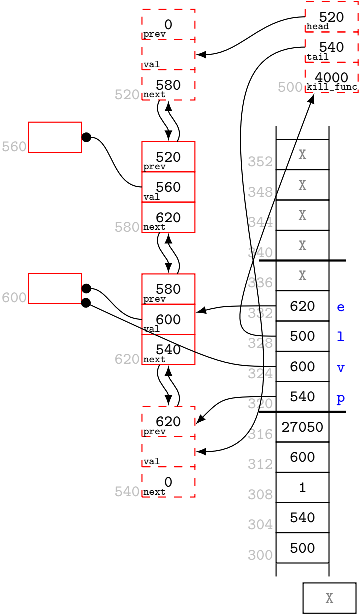
Exempel:

```c
/**
 * list_insert() - Insert a new element with a given valu
 * @l: List to manipulate.
 * @v: Value (pointer) to be inserted into the list.
 * @p: Position in the list before which the value should
 *
 * Creates a new element and inserts it into the list bef
 * Stores data in the new element.
 *
 * Returns: The position of the newly created element.
 */
list_pos list_insert(list *l, void *v, const list_pos p)
{
    // Allocate memory for a new cell.
    list_pos e = malloc(sizeof(cell));

    // Store the value.
    e->val = v;
    // Add links to/from the new cell.
    e->next = p;
    e->prev = p->prev;
    e->prev->next = e;
    p->prev = e;

    // Return the position of the new cell.
    return e;
}
```



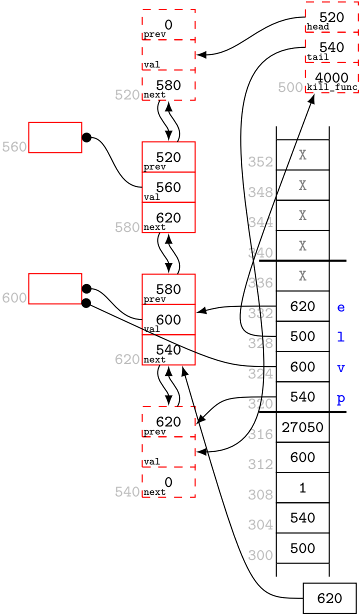
Exempel:

```c
/**
 * list_insert() - Insert a new element with a given valu
 * @l: List to manipulate.
 * @v: Value (pointer) to be inserted into the list.
 * @p: Position in the list before which the value should
 *
 * Creates a new element and inserts it into the list bef
 * Stores data in the new element.
 *
 * Returns: The position of the newly created element.
 */
list_pos list_insert(list *l, void *v, const list_pos p)
{
    // Allocate memory for a new cell.
    list_pos e = malloc(sizeof(cell));

    // Store the value.
    e->val = v;
    // Add links to/from the new cell.
    e->next = p;
    e->prev = p->prev;
    e->prev->next = e;
    p->prev = e;

    // Return the position of the new cell.
    return e;
}
```



Exempel:

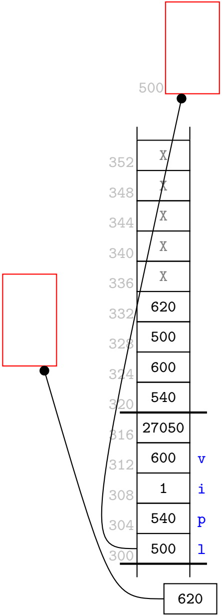
```c
/**
 * list_insert() - Insert a new element with a given valu
 * @l: List to manipulate.
 * @v: Value (pointer) to be inserted into the list.
 * @p: Position in the list before which the value should
 *
 * Creates a new element and inserts it into the list bef
 * Stores data in the new element.
 *
 * Returns: The position of the newly created element.
 */
list_pos list_insert(list *l, void *v, const list_pos p)
{
    // Allocate memory for a new cell.
    list_pos e = malloc(sizeof(cell));

    // Store the value.
    e->val = v;
    // Add links to/from the new cell.
    e->next = p;
    e->prev = p->prev;
    e->prev->next = e;
    p->prev = e;

    // Return the position of the new cell.
    return e;
}
```



[redacted]

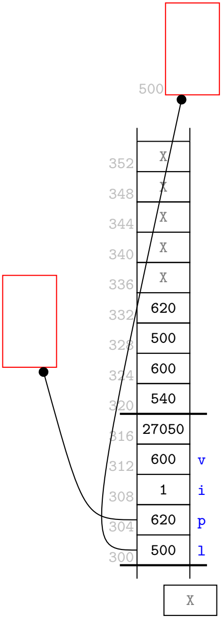
Exempel:

```c
int *int_create(int v)
{
    int *p = malloc(sizeof(*p)); // allocate payload
    *p = v; // assign value
    return p;
}
void int_kill(void *v)
{
    free(v); // deallocate payload
}
int main(void)
{
    // Create empty list, hand over responsibility to
    // deallocate payload using int_kill
    list *l = list_empty(int_kill);
    list_pos p = list_first(l);
    for (int i = 0; i < 2; i++) {
        int *v = int_create(20 + i);
        p = list_insert(l, v, p);
        p = list_next(l, p);
    }
    list_print(l, print_int);






    // Clean up the list, including payload
    list_kill(l);
    return 0;
}
```



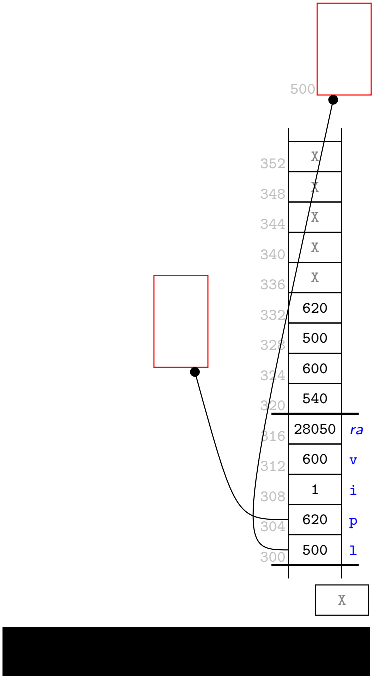
Exempel:

```c
int *int_create(int v)
{
    int *p = malloc(sizeof(*p)); // allocate payload
    *p = v; // assign value
    return p;
}
void int_kill(void *v)
{
    free(v); // deallocate payload
}
int main(void)
{
    // Create empty list, hand over responsibility to
    // deallocate payload using int_kill
    list *l = list_empty(int_kill);
    list_pos p = list_first(l);
    for (int i = 0; i < 2; i++) {
        int *v = int_create(20 + i);
        p = list_insert(l, v, p);
        p = list_next(l, p);
    }
    list_print(l, print_int);






    // Clean up the list, including payload
    list_kill(l);
    return 0;
}
```



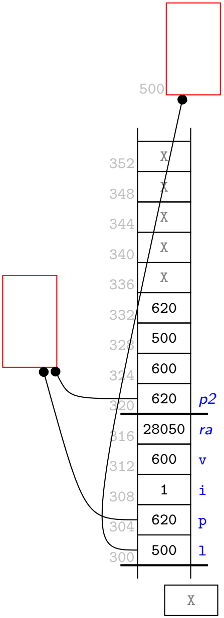
Exempel:

```c
int *int_create(int v)
{
    int *p = malloc(sizeof(*p)); // allocate payload
    *p = v; // assign value
    return p;
}
void int_kill(void *v)
{
    free(v); // deallocate payload
}
int main(void)
{
    // Create empty list, hand over responsibility to
    // deallocate payload using int_kill
    list *l = list_empty(int_kill);
    list_pos p = list_first(l);
    for (int i = 0; i < 2; i++) {
        int *v = int_create(20 + i);
        p = list_insert(l, v, p);
        p = list_next(l, p);
    }
    list_print(l, print_int);






    // Clean up the list, including payload
    list_kill(l);
    return 0;
}
```



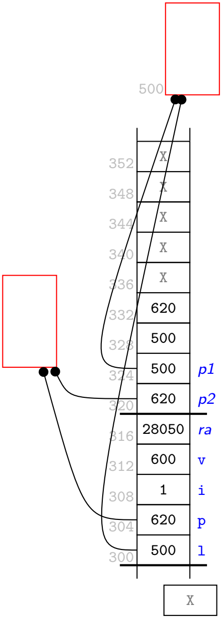
Exempel:

```c
int *int_create(int v)
{
    int *p = malloc(sizeof(*p)); // allocate payload
    *p = v; // assign value
    return p;
}
void int_kill(void *v)
{
    free(v); // deallocate payload
}
int main(void)
{
    // Create empty list, hand over responsibility to
    // deallocate payload using int_kill
    list *l = list_empty(int_kill);
    list_pos p = list_first(l);
    for (int i = 0; i < 2; i++) {
        int *v = int_create(20 + i);
        p = list_insert(l, v, p);
        p = list_next(l, p);
    }
    list_print(l, print_int);





    // Clean up the list, including payload
    list_kill(l);
    return 0;
}
```



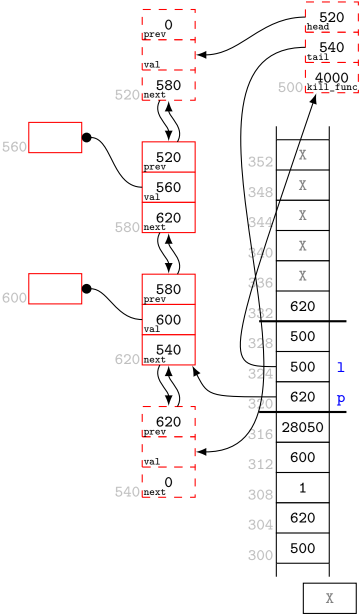
Exempel:

```
int *int_create(int v)
{
    int *p = malloc(sizeof(*p)); // allocate payload
    *p = v; // assign value
    return p;
}
void int_kill(void *v)
{
    free(v); // deallocate payload
}
int main(void)
{
    // Create empty list, hand over responsibility to
    // deallocate payload using int_kill
    list *l = list_empty(int_kill);
    list_pos p = list_first(l);
    for (int i = 0; i < 2; i++) {
        int *v = int_create(20 + i);
        p = list_insert(l, v, p);
        p = list_next(l, p);
    }
    list_print(l, print_int);






    // Clean up the list, including payload
    list_kill(l);
    return 0;
}
```



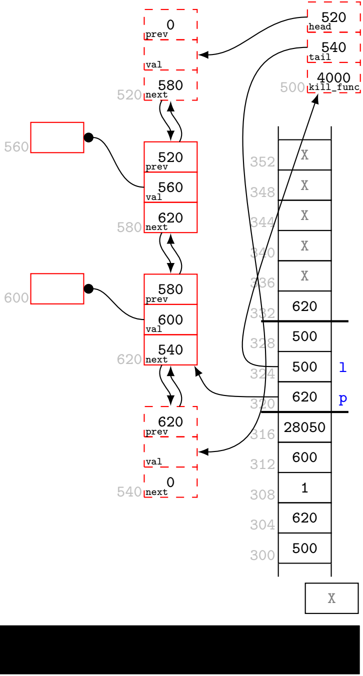
Exempel:

```c
/**
 * list_next() - Return the next position in a list.
 * @l: List to inspect.
 * @p: Any valid position except the last in the list.
 *
 * Returns: The position in the list after the given posi
 *          NOTE: The return value is undefined for the l
 */
list_pos list_next(const list *l, const list_pos p)
{
    return p->next;
}
```



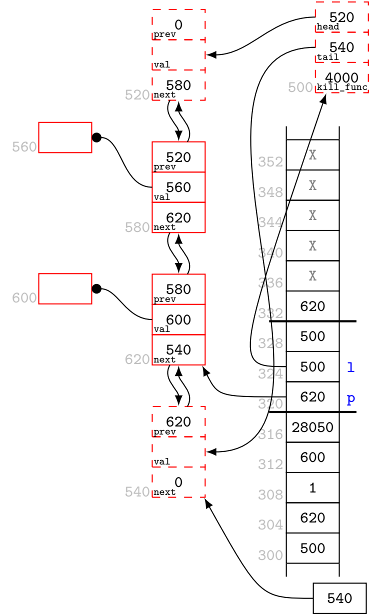
Exempel:

```c
/**
 * list_next() - Return the next position in a list.
 * @l: List to inspect.
 * @p: Any valid position except the last in the list.
 *
 * Returns: The position in the list after the given posi
 *          NOTE: The return value is undefined for the l
 */
list_pos list_next(const list *l, const list_pos p)
{
    return p->next;
}
```



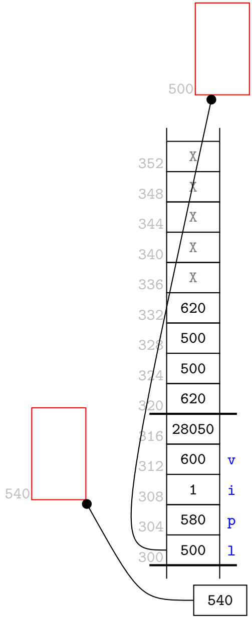
Exempel:

```c
/**
 * list_next() - Return the next position in a list.
 * @l: List to inspect.
 * @p: Any valid position except the last in the list.
 *
 * Returns: The position in the list after the given posi
 *          NOTE: The return value is undefined for the l
 */
list_pos list_next(const list *l, const list_pos p)
{
    return p->next;
}
```



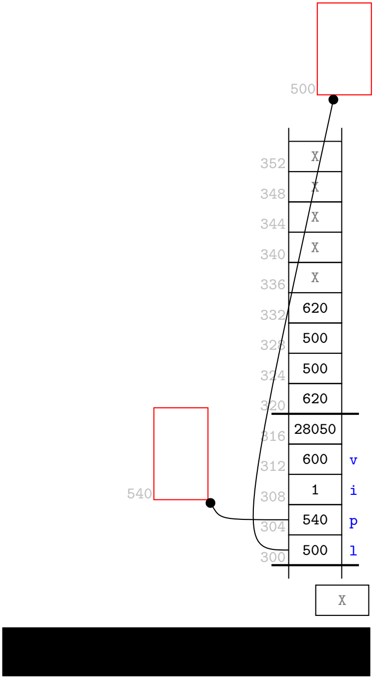
Exempel:

```c
int *int_create(int v)
{
    int *p = malloc(sizeof(*p)); // allocate payload
    *p = v; // assign value
    return p;
}
void int_kill(void *v)
{
    free(v); // deallocate payload
}
int main(void)
{
    // Create empty list, hand over responsibility to
    // deallocate payload using int_kill
    list *l = list_empty(int_kill);
    list_pos p = list_first(l);
    for (int i = 0; i < 2; i++) {
        int *v = int_create(20 + i);
        p = list_insert(l, v, p);
        p = list_next(l, p);
    }
    list_print(l, print_int);




    // Clean up the list, including payload
    list_kill(l);
    return 0;
}
```



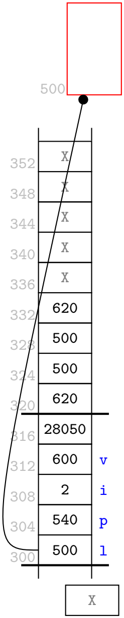
Exempel:

```c
int *int_create(int v)
{
    int *p = malloc(sizeof(*p)); // allocate payload
    *p = v; // assign value
    return p;
}
void int_kill(void *v)
{
    free(v); // deallocate payload
}
int main(void)
{
    // Create empty list, hand over responsibility to
    // deallocate payload using int_kill
    list *l = list_empty(int_kill);
    list_pos p = list_first(l);
    for (int i = 0; i < 2; i++) {
        int *v = int_create(20 + i);
        p = list_insert(l, v, p);
        p = list_next(l, p);
    }
    list_print(l, print_int);





    // Clean up the list, including payload
    list_kill(l);
    return 0;
}
```



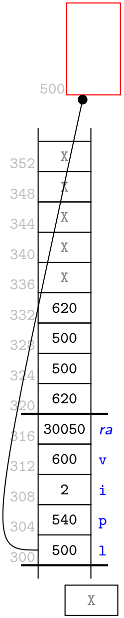
Exempel:

```c
int *int_create(int v)
{
    int *p = malloc(sizeof(*p)); // allocate payload
    *p = v; // assign value
    return p;
}
void int_kill(void *v)
{
    free(v); // deallocate payload
}
int main(void)
{
    // Create empty list, hand over responsibility to
    // deallocate payload using int_kill
    list *l = list_empty(int_kill);
    list_pos p = list_first(l);
    for (int i = 0; i < 2; i++) {
        int *v = int_create(20 + i);
        p = list_insert(l, v, p);
        p = list_next(l, p);
    }
    list_print(l, print_int);






    // Clean up the list, including payload
    list_kill(l);
    return 0;
}
```



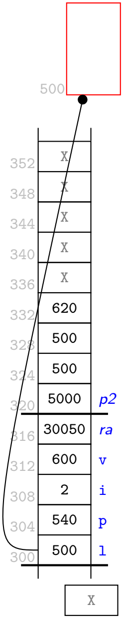
Exempel:

```c
int *int_create(int v)
{
    int *p = malloc(sizeof(*p)); // allocate payload
    *p = v; // assign value
    return p;
}
void int_kill(void *v)
{
    free(v); // deallocate payload
}
int main(void)
{
    // Create empty list, hand over responsibility to
    // deallocate payload using int_kill
    list *l = list_empty(int_kill);
    list_pos p = list_first(l);
    for (int i = 0; i < 2; i++) {
        int *v = int_create(20 + i);
        p = list_insert(l, v, p);
        p = list_next(l, p);
    }
    list_print(l, print_int);





    // Clean up the list, including payload
    list_kill(l);
    return 0;
}
```



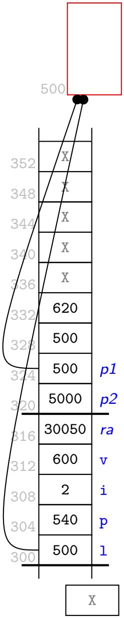
Exempel:

```c
int *int_create(int v)
{
    int *p = malloc(sizeof(*p)); // allocate payload
    *p = v; // assign value
    return p;
}
void int_kill(void *v)
{
    free(v); // deallocate payload
}
int main(void)
{
    // Create empty list, hand over responsibility to
    // deallocate payload using int_kill
    list *l = list_empty(int_kill);
    list_pos p = list_first(l);
    for (int i = 0; i < 2; i++) {
        int *v = int_create(20 + i);
        p = list_insert(l, v, p);
        p = list_next(l, p);
    }
    list_print(l, print_int);






    // Clean up the list, including payload
    list_kill(l);
    return 0;
}
```



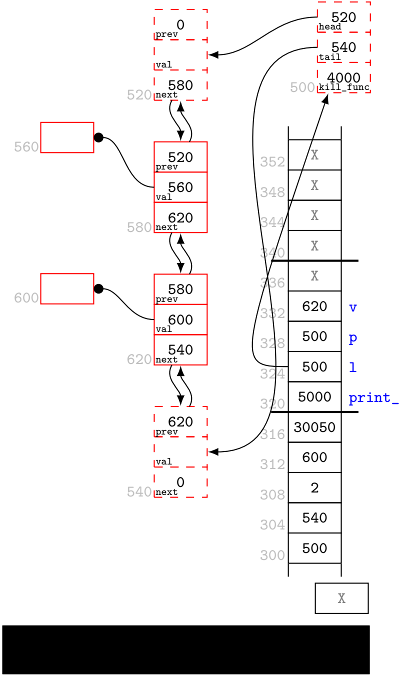
Exempel:

```c
int *int_create(int v)
{
    int *p = malloc(sizeof(*p)); // allocate payload
    *p = v; // assign value
    return p;
}
void int_kill(void *v)
{
    free(v); // deallocate payload
}
int main(void)
{
    // Create empty list, hand over responsibility to
    // deallocate payload using int_kill
    list *l = list_empty(int_kill);
    list_pos p = list_first(l);
    for (int i = 0; i < 2; i++) {
        int *v = int_create(20 + i);
        p = list_insert(l, v, p);
        p = list_next(l, p);
    }
    list_print(l, print_int);







    // Clean up the list, including payload
    list_kill(l);
    return 0;
}
```



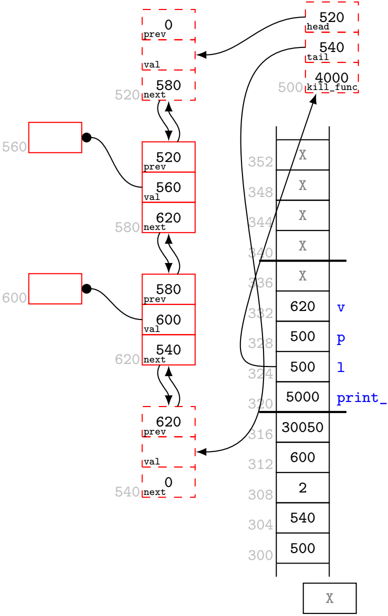
Exempel:

```c

/**
 * list_print() - Iterate over the list element and print
 * @l: List to inspect.
 * @print_func: Function called for each element.
 *
 * Iterates over the list and calls print_func with the v
 * in each element.
 *
 * Returns: Nothing.
 */
void list_print(const list *l, inspect_callback print_fun
{
    // Start at the beginning of the list.
    list_pos p = list_first(l);

    printf("( ");
    while (!list_pos_is_equal(l, p, list_end(l))) {
        // Call print_func with the element value at the
        // current position.
        void *v=list_inspect(l, p);
        print_func(v);
        // Advance to next position.
        p = list_next(l, p);
    }
    printf(" )\n");
}
```



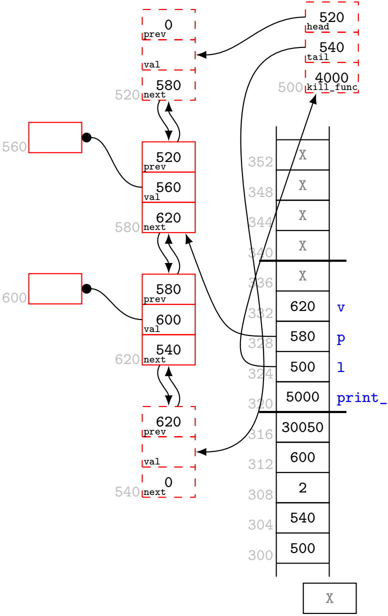
Exempel:

```c

/**
 * list_print() - Iterate over the list element and print
 * @l: List to inspect.
 * @print_func: Function called for each element.
 *
 * Iterates over the list and calls print_func with the v
 * in each element.
 *
 * Returns: Nothing.
 */
void list_print(const list *l, inspect_callback print_fun
{
    // Start at the beginning of the list.
    list_pos p = list_first(l);

    printf("( ");
    while (!list_pos_is_equal(l, p, list_end(l))) {
        // Call print_func with the element value at the
        // current position.
        void *v=list_inspect(l, p);
        print_func(v);
        // Advance to next position.
        p = list_next(l, p);
    }
    printf(" )\n");
}
```



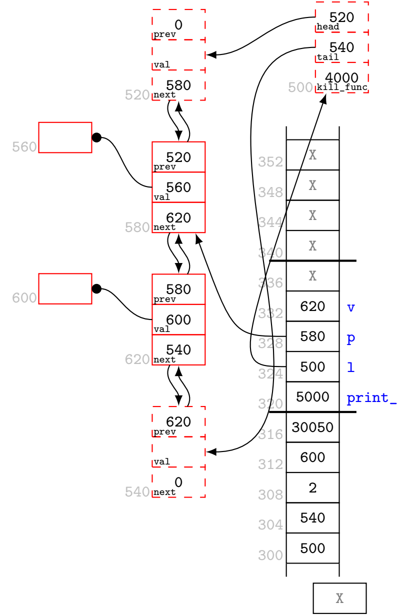
Exempel:

```c

/**
 * list_print() - Iterate over the list element and print
 * @l: List to inspect.
 * @print_func: Function called for each element.
 *
 * Iterates over the list and calls print_func with the v
 * in each element.
 *
 * Returns: Nothing.
 */
void list_print(const list *l, inspect_callback print_fun
{
    // Start at the beginning of the list.
    list_pos p = list_first(l);

    printf("( ");
    while (!list_pos_is_equal(l, p, list_end(l))) {
        // Call print_func with the element value at the
        // current position.
        void *v=list_inspect(l, p);
        print_func(v);
        // Advance to next position.
        p = list_next(l, p);
    }
    printf(" )\n");
}
```



Exempel:

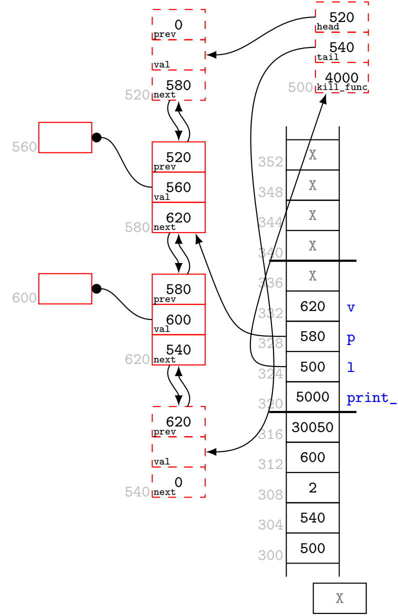
```c

/**
 * list_print() - Iterate over the list element and print
 * @l: List to inspect.
 * @print_func: Function called for each element.
 *
 * Iterates over the list and calls print_func with the v
 * in each element.
 *
 * Returns: Nothing.
 */
void list_print(const list *l, inspect_callback print_fun
{
    // Start at the beginning of the list.
    list_pos p = list_first(l);

    printf("( ");
    while (!list_pos_is_equal(l, p, list_end(l))) {
        // Call print_func with the element value at the
        // current position.
        void *v=list_inspect(l, p);
        print_func(v);
        // Advance to next position.
        p = list_next(l, p);
    }
    printf(" )\n");
}
```




```
(
```

Exempel:

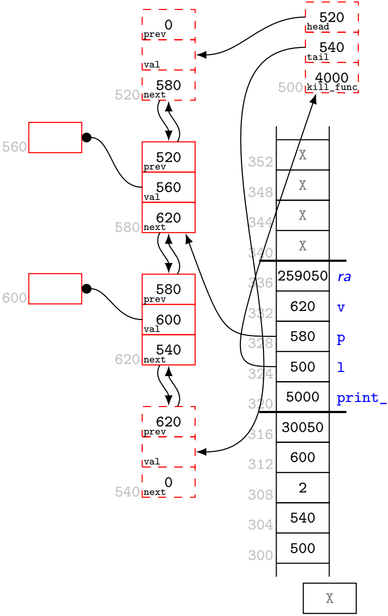
```c

/**
 * list_print() - Iterate over the list element and print
 * @l: List to inspect.
 * @print_func: Function called for each element.
 *
 * Iterates over the list and calls print_func with the v
 * in each element.
 *
 * Returns: Nothing.
 */
void list_print(const list *l, inspect_callback print_fun
{
    // Start at the beginning of the list.
    list_pos p = list_first(l);

    printf("( ");
    while (!list_pos_is_equal(l, p, list_end(l))) {
        // Call print_func with the element value at the
        // current position.
        void *v=list_inspect(l, p);
        print_func(v);
        // Advance to next position.
        p = list_next(l, p);
    }
    printf(" )\n");
}
```



(

Exempel:

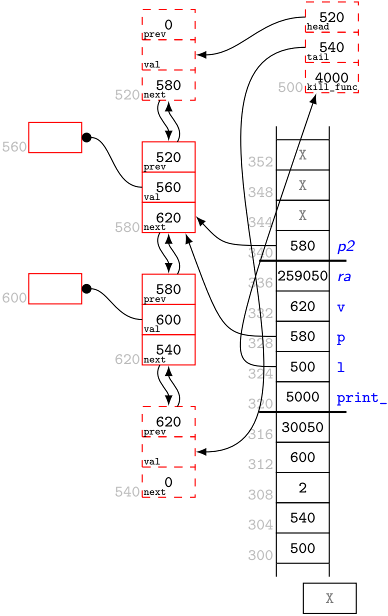
```c

/**
 * list_print() - Iterate over the list element and print
 * @l: List to inspect.
 * @print_func: Function called for each element.
 *
 * Iterates over the list and calls print_func with the v
 * in each element.
 *
 * Returns: Nothing.
 */
void list_print(const list *l, inspect_callback print_fun
{
    // Start at the beginning of the list.
    list_pos p = list_first(l);

    printf("( ");
    while (!list_pos_is_equal(l, p, list_end(l))) {
        // Call print_func with the element value at the
        // current position.
        void *v=list_inspect(l, p);
        print_func(v);
        // Advance to next position.
        p = list_next(l, p);
    }
    printf(" )\n");
}
```




```
(
```

Exempel:

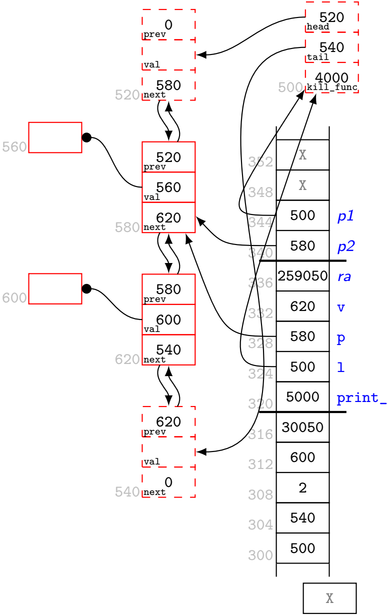
```c

/**
 * list_print() - Iterate over the list element and print
 * @l: List to inspect.
 * @print_func: Function called for each element.
 *
 * Iterates over the list and calls print_func with the v
 * in each element.
 *
 * Returns: Nothing.
 */
void list_print(const list *l, inspect_callback print_fun
{
    // Start at the beginning of the list.
    list_pos p = list_first(l);

    printf("( ");
    while (!list_pos_is_equal(l, p, list_end(l))) {
        // Call print_func with the element value at the
        // current position.
        void *v=list_inspect(l, p);
        print_func(v);
        // Advance to next position.
        p = list_next(l, p);
    }
    printf(" )\n");
}
```




```
(
```

Exempel:

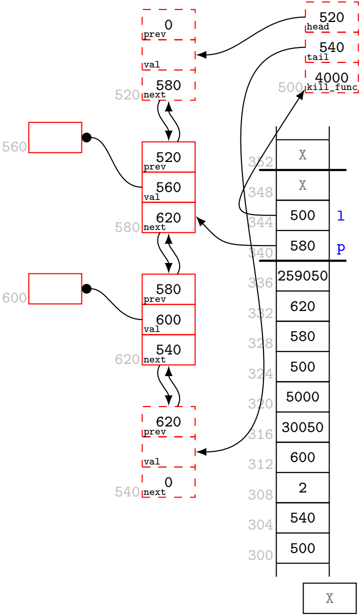
```c

/**
 * list_print() - Iterate over the list element and print
 * @l: List to inspect.
 * @print_func: Function called for each element.
 *
 * Iterates over the list and calls print_func with the v
 * in each element.
 *
 * Returns: Nothing.
 */
void list_print(const list *l, inspect_callback print_fun
{
    // Start at the beginning of the list.
    list_pos p = list_first(l);

    printf("( ");
    while (!list_pos_is_equal(l, p, list_end(l))) {
        // Call print_func with the element value at the
        // current position.
        void *v=list_inspect(l, p);
        print_func(v);
        // Advance to next position.
        p = list_next(l, p);
    }
    printf(" )\n");
}
```



(

Exempel:

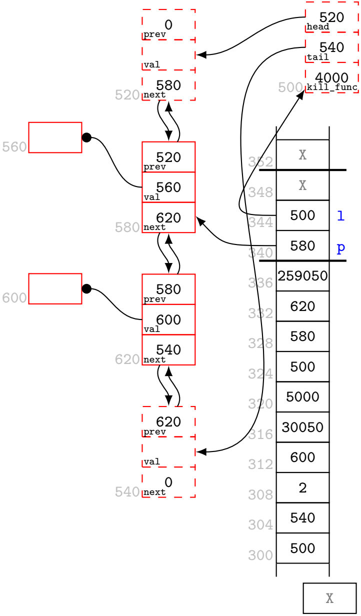
```c
/**
 * list_inspect() - Return the value of the element at a
 *                  position in a list.
 * @l: List to inspect.
 * @p: Any valid position in the list, except the last.
 *
 * Returns: Returns the value at the given position as a
 *          NOTE: The return value is undefined for the l
 */
void *list_inspect(const list *l, const list_pos p)
{
    return p->val;
}
```



(

Exempel:

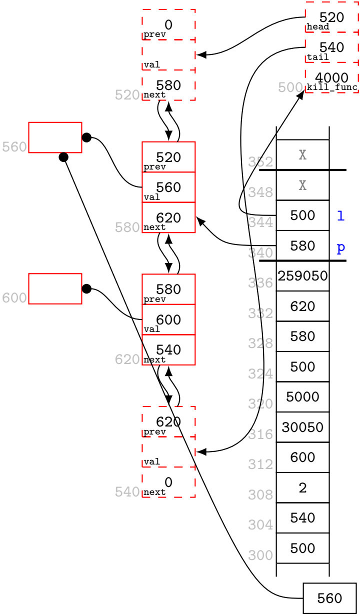
```c
/**
 * list_inspect() - Return the value of the element at a g
 *                  position in a list.
 * @l: List to inspect.
 * @p: Any valid position in the list, except the last.
 *
 * Returns: Returns the value at the given position as a v
 *          NOTE: The return value is undefined for the la
 */
void *list_inspect(const list *l, const list_pos p)
{
    return p->val;
}
```



(

Exempel:

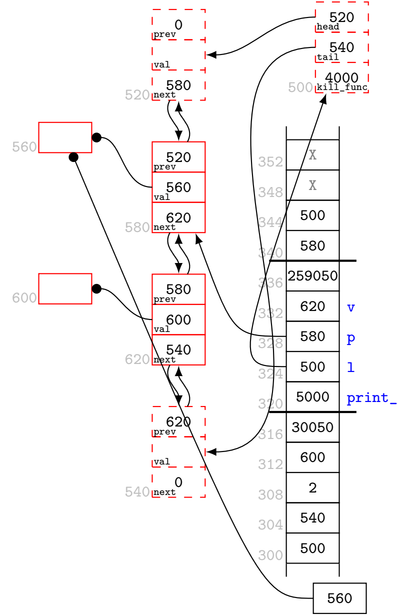
```c
/**
 * list_inspect() - Return the value of the element at a 
 *                  position in a list.
 * @l: List to inspect.
 * @p: Any valid position in the list, except the last.
 *
 * Returns: Returns the value at the given position as a 
 *          NOTE: The return value is undefined for the l
 */
void *list_inspect(const list *l, const list_pos p)
{
    return p->val;
}
```



(

Exempel:

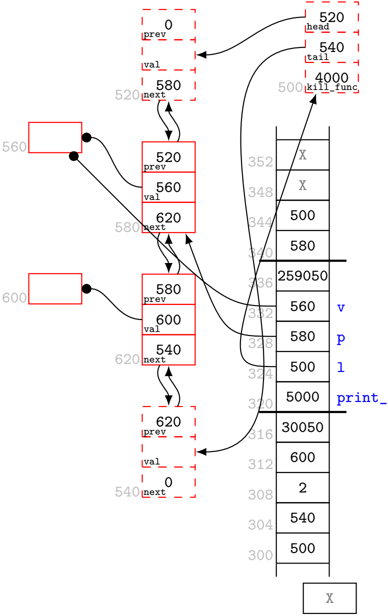
```
/**
 * list_print() - Iterate over the list element and print
 * @l: List to inspect.
 * @print_func: Function called for each element.
 *
 * Iterates over the list and calls print_func with the v
 * in each element.
 *
 * Returns: Nothing.
 */
void list_print(const list *l, inspect_callback print_fun
{
    // Start at the beginning of the list.
    list_pos p = list_first(l);

    printf("( ");
    while (!list_pos_is_equal(l, p, list_end(l))) {
        // Call print_func with the element value at the
        // current position.
        void *v=list_inspect(l, p);
        print_func(v);
        // Advance to next position.
        p = list_next(l, p);
    }
    printf(" )\n");
}
```



(

Exempel:

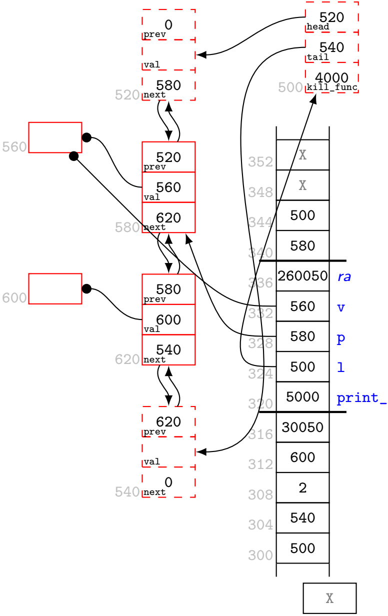
```c

/**
 * list_print() - Iterate over the list element and print
 * @l: List to inspect.
 * @print_func: Function called for each element.
 *
 * Iterates over the list and calls print_func with the v
 * in each element.
 *
 * Returns: Nothing.
 */
void list_print(const list *l, inspect_callback print_fun
{
    // Start at the beginning of the list.
    list_pos p = list_first(l);

    printf("( ");
    while (!list_pos_is_equal(l, p, list_end(l))) {
        // Call print_func with the element value at the
        // current position.
        void *v=list_inspect(l, p);
        print_func(v);
        // Advance to next position.
        p = list_next(l, p);
    }
    printf(" )\n");
}
```




```
(
```

Exempel:

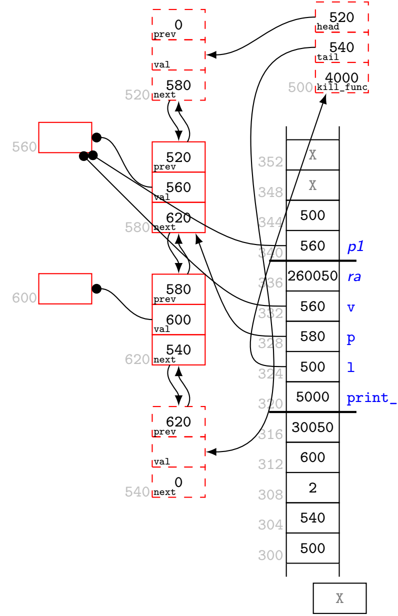
```c

/**
 * list_print() - Iterate over the list element and print
 * @l: List to inspect.
 * @print_func: Function called for each element.
 *
 * Iterates over the list and calls print_func with the v
 * in each element.
 *
 * Returns: Nothing.
 */
void list_print(const list *l, inspect_callback print_fun
{
    // Start at the beginning of the list.
    list_pos p = list_first(l);

    printf("( ");
    while (!list_pos_is_equal(l, p, list_end(l))) {
        // Call print_func with the element value at the
        // current position.
        void *v=list_inspect(l, p);
        print_func(v);
        // Advance to next position.
        p = list_next(l, p);
    }
    printf(" )\n");
}
```




```
(
```

Exempel:

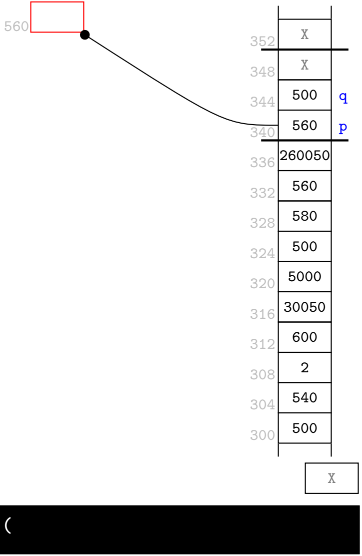
```c

/**
 * list_print() - Iterate over the list element and print
 * @l: List to inspect.
 * @print_func: Function called for each element.
 *
 * Iterates over the list and calls print_func with the v
 * in each element.
 *
 * Returns: Nothing.
 */
void list_print(const list *l, inspect_callback print_fun
{
    // Start at the beginning of the list.
    list_pos p = list_first(l);

    printf("( ");
    while (!list_pos_is_equal(l, p, list_end(l))) {
        // Call print_func with the element value at the
        // current position.
        void *v=list_inspect(l, p);
        print_func(v);
        // Advance to next position.
        p = list_next(l, p);
    }
    printf(" )\n");
}
```



(

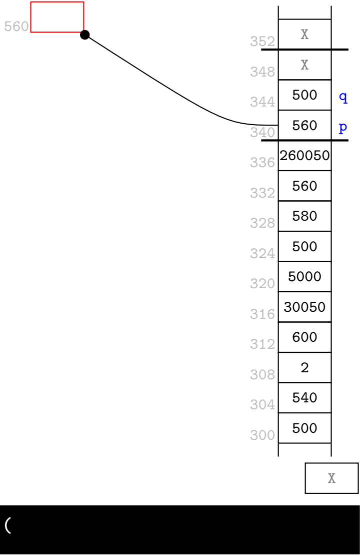
Exempel:

```
void print_int(const void *p)
{
    const int *q = p;
    printf("%d ", *q);
}
```



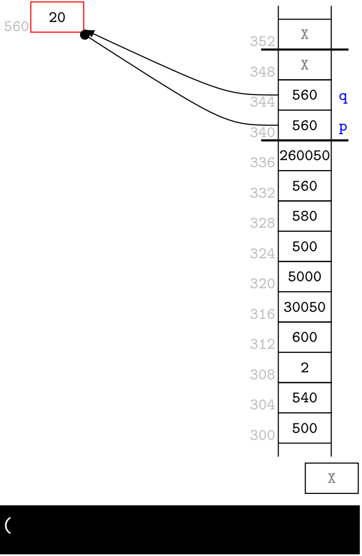
Exempel:

```c
void print_int(const void *p)
{
    const int *q = p;
    printf("%d ", *q);
}
```



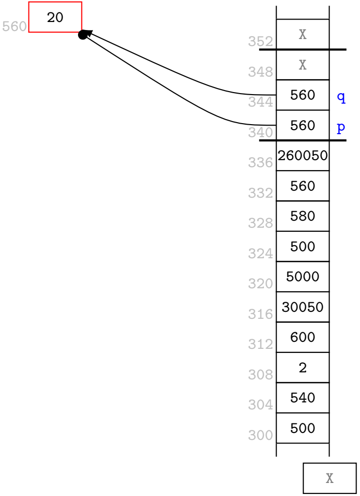
Exempel:

```c
void print_int(const void *p)
{
    const int *q = p;
    printf("%d ", *q);
}
```



Exempel:

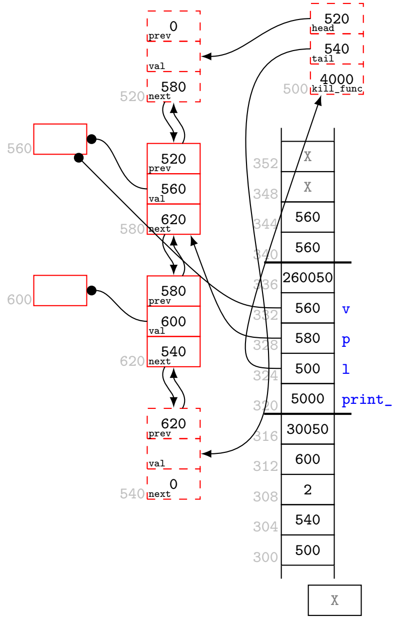
```c
void print_int(const void *p)
{
    const int *q = p;
    printf("%d ", *q);
}
```




```
( 20
```

Exempel:

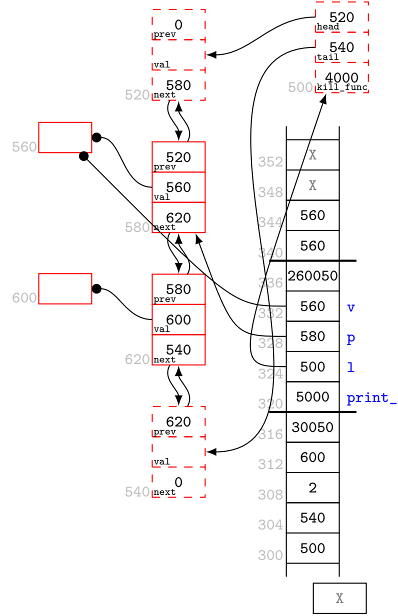
```c

/**
 * list_print() - Iterate over the list element and print
 * @l: List to inspect.
 * @print_func: Function called for each element.
 *
 * Iterates over the list and calls print_func with the v
 * in each element.
 *
 * Returns: Nothing.
 */
void list_print(const list *l, inspect_callback print_fun
{
    // Start at the beginning of the list.
    list_pos p = list_first(l);

    printf("( ");
    while (!list_pos_is_equal(l, p, list_end(l))) {
        // Call print_func with the element value at the
        // current position.
        void *v=list_inspect(l, p);
        print_func(v);
        // Advance to next position.
        p = list_next(l, p);
    }
    printf(" )\n");
}
```



(20

Exempel:

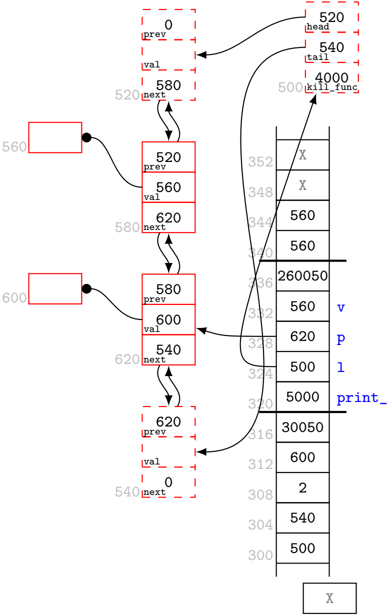
```c

/**
 * list_print() - Iterate over the list element and print
 * @l: List to inspect.
 * @print_func: Function called for each element.
 *
 * Iterates over the list and calls print_func with the v
 * in each element.
 *
 * Returns: Nothing.
 */
void list_print(const list *l, inspect_callback print_fun
{
    // Start at the beginning of the list.
    list_pos p = list_first(l);

    printf("( ");
    while (!list_pos_is_equal(l, p, list_end(l))) {
        // Call print_func with the element value at the
        // current position.
        void *v=list_inspect(l, p);
        print_func(v);
        // Advance to next position.
        p = list_next(l, p);
    }
    printf(" )\n");
}
```



(20

Exempel:

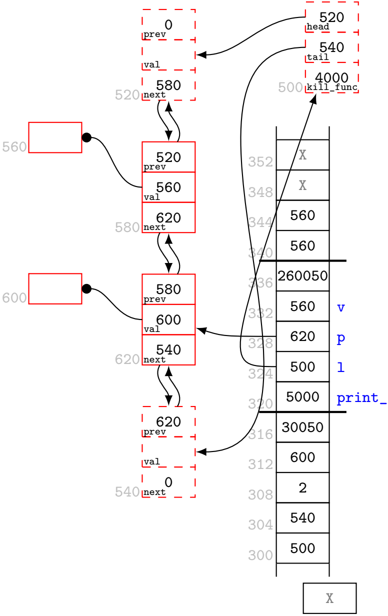
```
/**
 * list_print() - Iterate over the list element and print
 * @l: List to inspect.
 * @print_func: Function called for each element.
 *
 * Iterates over the list and calls print_func with the v
 * in each element.
 *
 * Returns: Nothing.
 */
void list_print(const list *l, inspect_callback print_fun
{
    // Start at the beginning of the list.
    list_pos p = list_first(l);

    printf("( ");
    while (!list_pos_is_equal(l, p, list_end(l))) {
        // Call print_func with the element value at the
        // current position.
        void *v=list_inspect(l, p);
        print_func(v);
        // Advance to next position.
        p = list_next(l, p);
    }
    printf(" )\n");
}
```




```
( 20
```

Exempel:

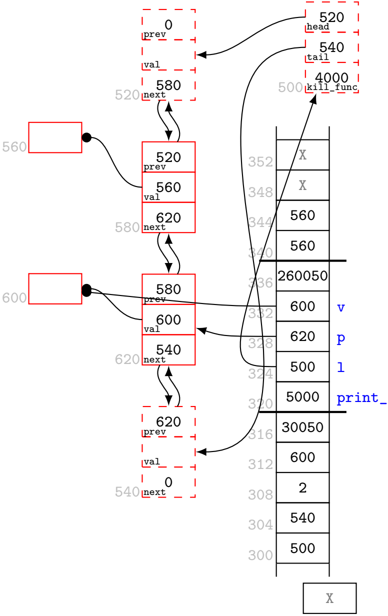
```c

/**
 * list_print() - Iterate over the list element and print
 * @l: List to inspect.
 * @print_func: Function called for each element.
 *
 * Iterates over the list and calls print_func with the v
 * in each element.
 *
 * Returns: Nothing.
 */
void list_print(const list *l, inspect_callback print_fun
{
    // Start at the beginning of the list.
    list_pos p = list_first(l);

    printf("( ");
    while (!list_pos_is_equal(l, p, list_end(l))) {
        // Call print_func with the element value at the
        // current position.
        void *v=list_inspect(l, p);
        print_func(v);
        // Advance to next position.
        p = list_next(l, p);
    }
    printf(" )\n");
}
```



(20

Exempel:

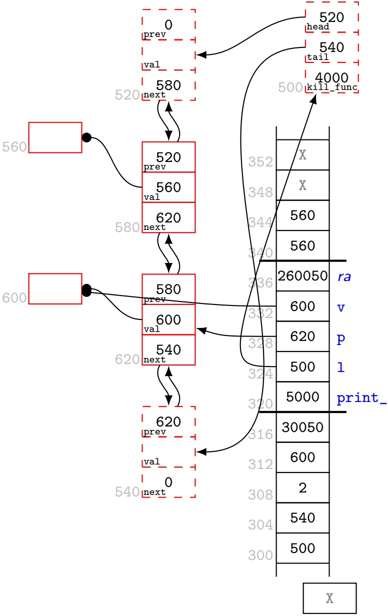
```c

/**
 * list_print() - Iterate over the list element and print
 * @l: List to inspect.
 * @print_func: Function called for each element.
 *
 * Iterates over the list and calls print_func with the v
 * in each element.
 *
 * Returns: Nothing.
 */
void list_print(const list *l, inspect_callback print_fun
{
    // Start at the beginning of the list.
    list_pos p = list_first(l);

    printf("( ");
    while (!list_pos_is_equal(l, p, list_end(l))) {
        // Call print_func with the element value at the
        // current position.
        void *v=list_inspect(l, p);
        print_func(v);
        // Advance to next position.
        p = list_next(l, p);
    }
    printf(" )\n");
}
```




```
( 20
```

Exempel:

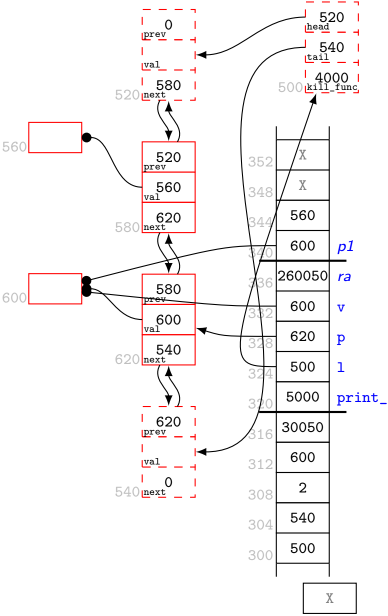
```c

/**
 * list_print() - Iterate over the list element and print
 * @l: List to inspect.
 * @print_func: Function called for each element.
 *
 * Iterates over the list and calls print_func with the v
 * in each element.
 *
 * Returns: Nothing.
 */
void list_print(const list *l, inspect_callback print_fun
{
    // Start at the beginning of the list.
    list_pos p = list_first(l);

    printf("( ");
    while (!list_pos_is_equal(l, p, list_end(l))) {
        // Call print_func with the element value at the
        // current position.
        void *v=list_inspect(l, p);
        print_func(v);
        // Advance to next position.
        p = list_next(l, p);
    }
    printf(" )\n");
}
```



(20

Exempel:

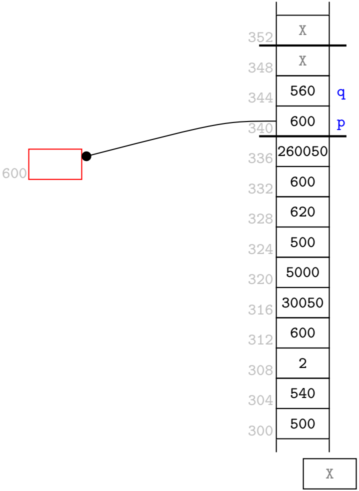
```
/**
 * list_print() - Iterate over the list element and print
 * @l: List to inspect.
 * @print_func: Function called for each element.
 *
 * Iterates over the list and calls print_func with the v
 * in each element.
 *
 * Returns: Nothing.
 */
void list_print(const list *l, inspect_callback print_fun
{
    // Start at the beginning of the list.
    list_pos p = list_first(l);

    printf("( ");
    while (!list_pos_is_equal(l, p, list_end(l))) {
        // Call print_func with the element value at the
        // current position.
        void *v=list_inspect(l, p);
        print_func(v);
        // Advance to next position.
        p = list_next(l, p);
    }
    printf(" )\n");
}
```




```
( 20
```

Exempel:

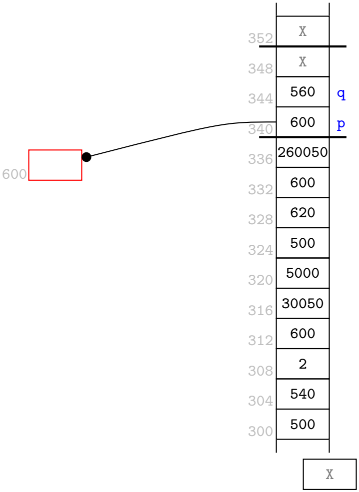
```c
void print_int(const void *p)
{
    const int *q = p;
    printf("%d ", *q);
}
```



(20

Exempel:

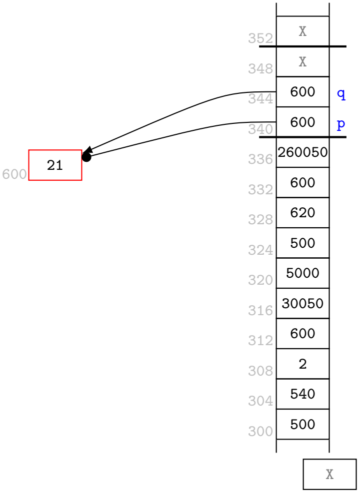
```c
void print_int(const void *p)
{
    const int *q = p;
    printf("%d ", *q);
}
```




```
( 20
```

Exempel:

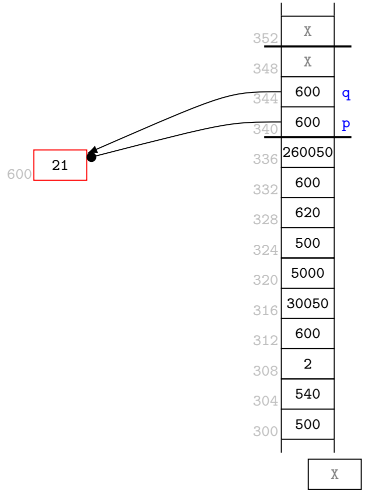
```c
void print_int(const void *p)
{
    const int *q = p;
    printf("%d ", *q);
}
```




```
( 20
```

Exempel:

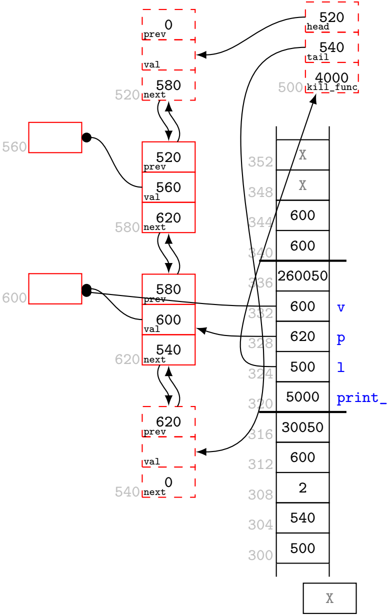
```
void print_int(const void *p)
{
    const int *q = p;
    printf("%d ", *q);
}
```



(20 21

Exempel:

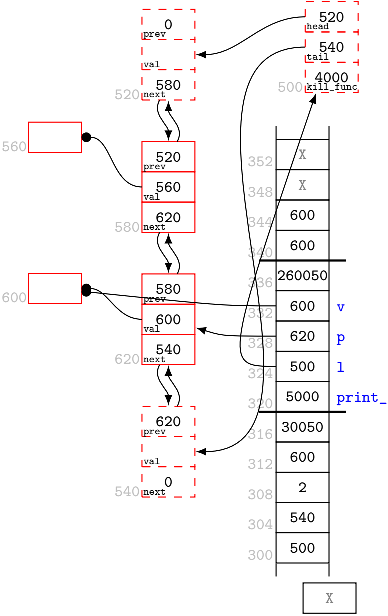
```c

/**
 * list_print() - Iterate over the list element and print
 * @l: List to inspect.
 * @print_func: Function called for each element.
 *
 * Iterates over the list and calls print_func with the v
 * in each element.
 *
 * Returns: Nothing.
 */
void list_print(const list *l, inspect_callback print_fun
{
    // Start at the beginning of the list.
    list_pos p = list_first(l);

    printf("( ");
    while (!list_pos_is_equal(l, p, list_end(l))) {
        // Call print_func with the element value at the
        // current position.
        void *v=list_inspect(l, p);
        print_func(v);
        // Advance to next position.
        p = list_next(l, p);
    }
    printf(" )\n");
}
```



(20 21

Exempel:

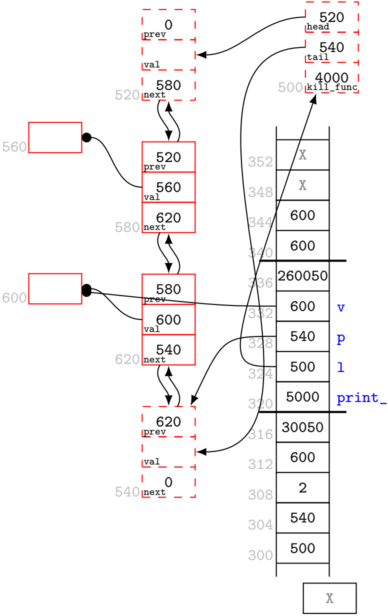
```c

/**
 * list_print() - Iterate over the list element and print
 * @l: List to inspect.
 * @print_func: Function called for each element.
 *
 * Iterates over the list and calls print_func with the v
 * in each element.
 *
 * Returns: Nothing.
 */
void list_print(const list *l, inspect_callback print_fun
{
    // Start at the beginning of the list.
    list_pos p = list_first(l);

    printf("( ");
    while (!list_pos_is_equal(l, p, list_end(l))) {
        // Call print_func with the element value at the
        // current position.
        void *v=list_inspect(l, p);
        print_func(v);
        // Advance to next position.
        p = list_next(l, p);
    }
    printf(" )\n");
}
```



(20 21

Exempel:

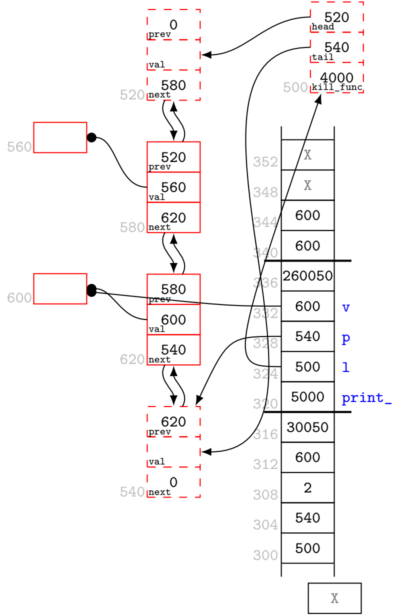
```c

/**
 * list_print() - Iterate over the list element and print
 * @l: List to inspect.
 * @print_func: Function called for each element.
 *
 * Iterates over the list and calls print_func with the v
 * in each element.
 *
 * Returns: Nothing.
 */
void list_print(const list *l, inspect_callback print_fun
{
    // Start at the beginning of the list.
    list_pos p = list_first(l);

    printf("( ");
    while (!list_pos_is_equal(l, p, list_end(l))) {
        // Call print_func with the element value at the
        // current position.
        void *v=list_inspect(l, p);
        print_func(v);
        // Advance to next position.
        p = list_next(l, p);
    }
    printf(" )\n");
}
```



(20 21

Exempel:

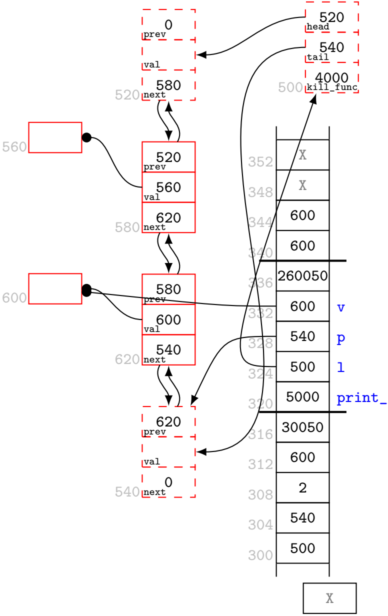
```c

/**
 * list_print() - Iterate over the list element and print
 * @l: List to inspect.
 * @print_func: Function called for each element.
 *
 * Iterates over the list and calls print_func with the v
 * in each element.
 *
 * Returns: Nothing.
 */
void list_print(const list *l, inspect_callback print_fun
{
    // Start at the beginning of the list.
    list_pos p = list_first(l);

    printf("( ");
    while (!list_pos_is_equal(l, p, list_end(l))) {
        // Call print_func with the element value at the
        // current position.
        void *v=list_inspect(l, p);
        print_func(v);
        // Advance to next position.
        p = list_next(l, p);
    }
    printf(" )\n");
}
```




```
( 20 21
```

Exempel:

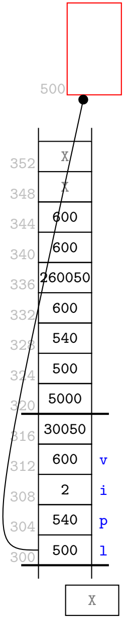
```c

/**
 * list_print() - Iterate over the list element and print
 * @l: List to inspect.
 * @print_func: Function called for each element.
 *
 * Iterates over the list and calls print_func with the v
 * in each element.
 *
 * Returns: Nothing.
 */
void list_print(const list *l, inspect_callback print_fun
{
    // Start at the beginning of the list.
    list_pos p = list_first(l);

    printf("( ");
    while (!list_pos_is_equal(l, p, list_end(l))) {
        // Call print_func with the element value at the
        // current position.
        void *v=list_inspect(l, p);
        print_func(v);
        // Advance to next position.
        p = list_next(l, p);
    }
    printf(" )\n");
}
```




```
( 20 21 )
```

Exempel:

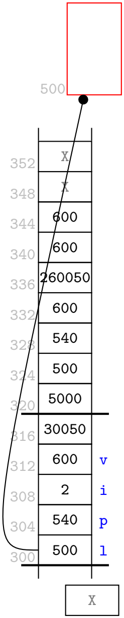
```c
int *int_create(int v)
{
    int *p = malloc(sizeof(*p)); // allocate payload
    *p = v; // assign value
    return p;
}
void int_kill(void *v)
{
    free(v); // deallocate payload
}
int main(void)
{
    // Create empty list, hand over responsibility to
    // deallocate payload using int_kill
    list *l = list_empty(int_kill);
    list_pos p = list_first(l);
    for (int i = 0; i < 2; i++) {
        int *v = int_create(20 + i);
        p = list_insert(l, v, p);
        p = list_next(l, p);
    }
    list_print(l, print_int);






    // Clean up the list, including payload
    list_kill(l);
    return 0;
}
```



(20 21)

Exempel:

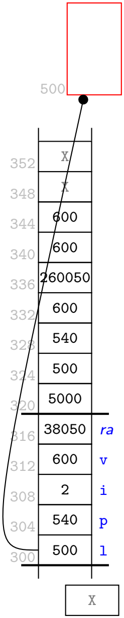
```c
int *int_create(int v)
{
    int *p = malloc(sizeof(*p)); // allocate payload
    *p = v; // assign value
    return p;
}
void int_kill(void *v)
{
    free(v); // deallocate payload
}
int main(void)
{
    // Create empty list, hand over responsibility to
    // deallocate payload using int_kill
    list *l = list_empty(int_kill);
    list_pos p = list_first(l);
    for (int i = 0; i < 2; i++) {
        int *v = int_create(20 + i);
        p = list_insert(l, v, p);
        p = list_next(l, p);
    }
    list_print(l, print_int);






    // Clean up the list, including payload
    list_kill(l);
    return 0;
}
```



(20 21)

Exempel:

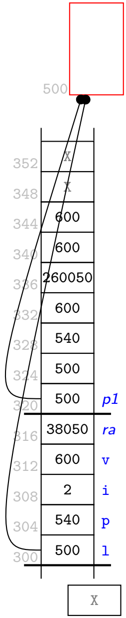
```c
int *int_create(int v)
{
    int *p = malloc(sizeof(*p)); // allocate payload
    *p = v; // assign value
    return p;
}
void int_kill(void *v)
{
    free(v); // deallocate payload
}
int main(void)
{
    // Create empty list, hand over responsibility to
    // deallocate payload using int_kill
    list *l = list_empty(int_kill);
    list_pos p = list_first(l);
    for (int i = 0; i < 2; i++) {
        int *v = int_create(20 + i);
        p = list_insert(l, v, p);
        p = list_next(l, p);
    }
    list_print(l, print_int);






    // Clean up the list, including payload
    list_kill(l);
    return 0;
}
```



(20 21)

Exempel:

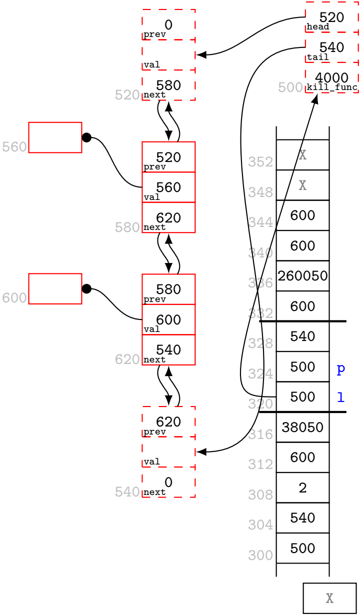
```c
int *int_create(int v)
{
    int *p = malloc(sizeof(*p)); // allocate payload
    *p = v; // assign value
    return p;
}
void int_kill(void *v)
{
    free(v); // deallocate payload
}
int main(void)
{
    // Create empty list, hand over responsibility to
    // deallocate payload using int_kill
    list *l = list_empty(int_kill);
    list_pos p = list_first(l);
    for (int i = 0; i < 2; i++) {
        int *v = int_create(20 + i);
        p = list_insert(l, v, p);
        p = list_next(l, p);
    }
    list_print(l, print_int);






    // Clean up the list, including payload
    list_kill(l);
    return 0;
}
```




```
( 20 21 )
```

Exempel:

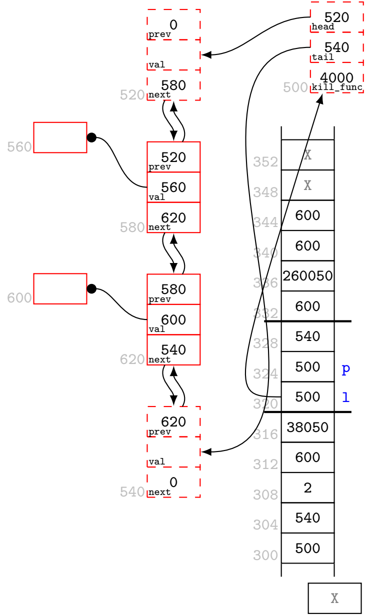
```c
/**
 * list_kill() - Destroy a given list.
 * @l: List to destroy.
 *
 * Returns all dynamic memory used by the list and its el
 * kill_func was registered at list creation, also calls
 * element to return any user-allocated memory occupied b
 * values.
 *
 * Returns: Nothing.
 */
void list_kill(list *l)
{
    // Use public functions to traverse the list.

    // Start with the first element (will be defined
    // even for an empty list).
    list_pos p = list_first(l);

    // Remove first element until list is empty.
    while (!list_is_empty(l)) {
        p = list_remove(l, p);
    }

    // Free border elements and the list head.
    free(l->head);
    free(l->tail);
    free(l);
}
```



(20 21)

Exempel:

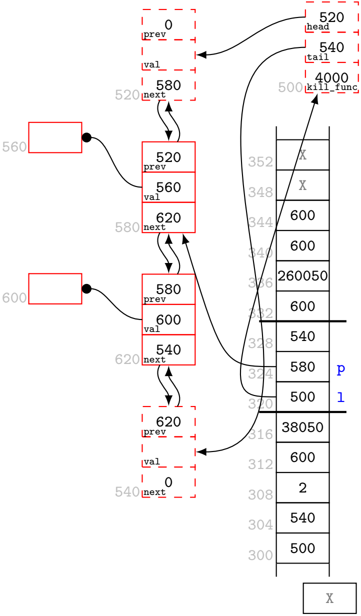
```c
/**
 * list_kill() - Destroy a given list.
 * @l: List to destroy.
 *
 * Returns all dynamic memory used by the list and its el
 * kill_func was registered at list creation, also calls 
 * element to return any user-allocated memory occupied b
 * values.
 *
 * Returns: Nothing.
 */
void list_kill(list *l)
{
    // Use public functions to traverse the list.

    // Start with the first element (will be defined
    // even for an empty list).
    list_pos p = list_first(l);

    // Remove first element until list is empty.
    while (!list_is_empty(l)) {
        p = list_remove(l, p);
    }

    // Free border elements and the list head.
    free(l->head);
    free(l->tail);
    free(l);
}
```




```
( 20 21 )
```

Exempel:

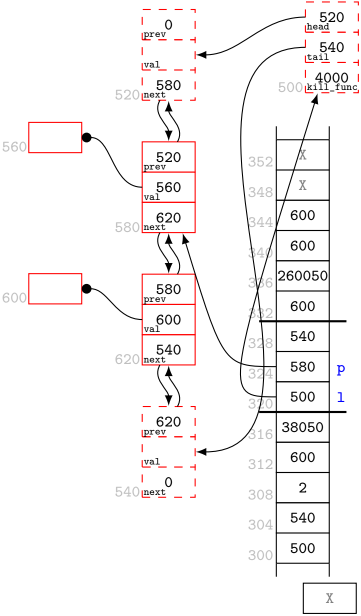
```c
/**
 * list_kill() - Destroy a given list.
 * @l: List to destroy.
 *
 * Returns all dynamic memory used by the list and its el
 * kill_func was registered at list creation, also calls
 * element to return any user-allocated memory occupied b
 * values.
 *
 * Returns: Nothing.
 */
void list_kill(list *l)
{
    // Use public functions to traverse the list.

    // Start with the first element (will be defined
    // even for an empty list).
    list_pos p = list_first(l);

    // Remove first element until list is empty.
    while (!list_is_empty(l)) {
        p = list_remove(l, p);
    }

    // Free border elements and the list head.
    free(l->head);
    free(l->tail);
    free(l);
}
```



(20 21)

Exempel:

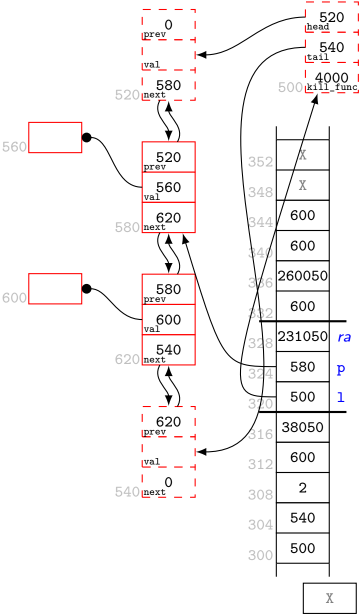
```c
/**
 * list_kill() - Destroy a given list.
 * @l: List to destroy.
 *
 * Returns all dynamic memory used by the list and its el
 * kill_func was registered at list creation, also calls
 * element to return any user-allocated memory occupied b
 * values.
 *
 * Returns: Nothing.
 */
void list_kill(list *l)
{
    // Use public functions to traverse the list.

    // Start with the first element (will be defined
    // even for an empty list).
    list_pos p = list_first(l);

    // Remove first element until list is empty.
    while (!list_is_empty(l)) {
        p = list_remove(l, p);
    }

    // Free border elements and the list head.
    free(l->head);
    free(l->tail);
    free(l);
}
```



(20 21)

Exempel:

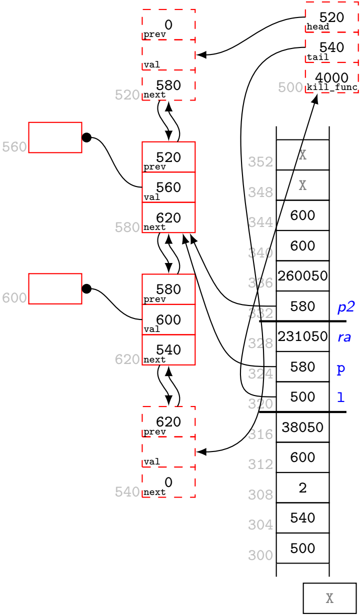
```c
/**
 * list_kill() - Destroy a given list.
 * @l: List to destroy.
 *
 * Returns all dynamic memory used by the list and its el
 * kill_func was registered at list creation, also calls
 * element to return any user-allocated memory occupied b
 * values.
 *
 * Returns: Nothing.
 */
void list_kill(list *l)
{
    // Use public functions to traverse the list.

    // Start with the first element (will be defined
    // even for an empty list).
    list_pos p = list_first(l);

    // Remove first element until list is empty.
    while (!list_is_empty(l)) {
        p = list_remove(l, p);
    }

    // Free border elements and the list head.
    free(l->head);
    free(l->tail);
    free(l);
}
```



(20 21)

Exempel:

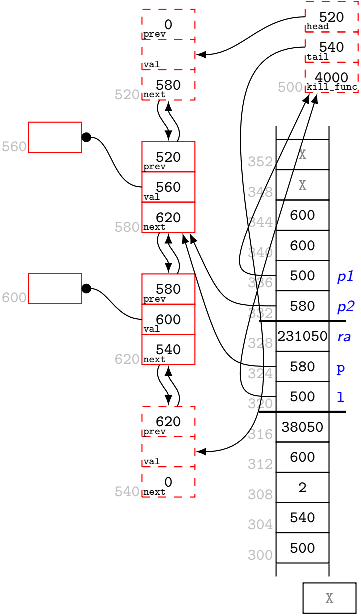
```c
/**
 * list_kill() - Destroy a given list.
 * @l: List to destroy.
 *
 * Returns all dynamic memory used by the list and its el
 * kill_func was registered at list creation, also calls
 * element to return any user-allocated memory occupied b
 * values.
 *
 * Returns: Nothing.
 */
void list_kill(list *l)
{
    // Use public functions to traverse the list.

    // Start with the first element (will be defined
    // even for an empty list).
    list_pos p = list_first(l);

    // Remove first element until list is empty.
    while (!list_is_empty(l)) {
        p = list_remove(l, p);
    }

    // Free border elements and the list head.
    free(l->head);
    free(l->tail);
    free(l);
}
```



(20 21)

Exempel:

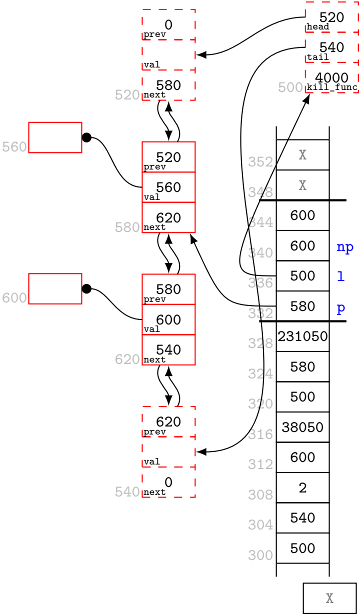
```c
/**
 * list_kill() - Destroy a given list.
 * @l: List to destroy.
 *
 * Returns all dynamic memory used by the list and its el
 * kill_func was registered at list creation, also calls 
 * element to return any user-allocated memory occupied b
 * values.
 *
 * Returns: Nothing.
 */
void list_kill(list *l)
{
    // Use public functions to traverse the list.

    // Start with the first element (will be defined
    // even for an empty list).
    list_pos p = list_first(l);

    // Remove first element until list is empty.
    while (!list_is_empty(l)) {
        p = list_remove(l, p);
    }

    // Free border elements and the list head.
    free(l->head);
    free(l->tail);
    free(l);
}
```



(20 21)

Exempel:

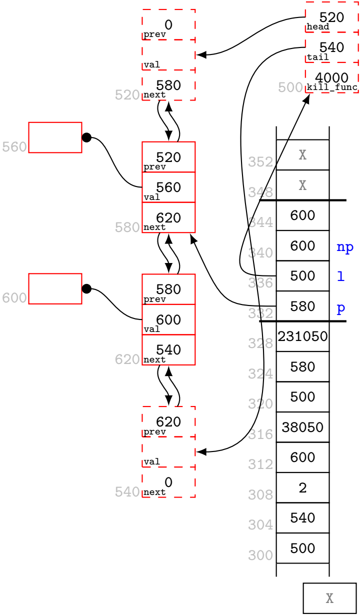
```c
/**
 * list_remove() - Remove an element from a list.
 * @l: List to manipulate.
 * @p: Position in the list of the element to remove.
 *
 * Removes the element at position p from the list. If a l
 * was registered at list creation, calls it to deallocat
 * held by the element value.
 *
 * Returns: The position after the removed element.
 */
list_pos list_remove(list *l, const list_pos p)
{
    // Remember return position.
    list_pos np = p->next;
    // Link past this element.
    p->prev->next = p->next;
    p->next->prev = p->prev;

    // Call kill_func if registered.
    if (l->kill_func != NULL) {
        // Return any user-allocated memory for the value
        l->kill_func(p->val);
    }
    // Return the memory allocated to the cell itself.
    free(p);
    // Return the position of the next element.
    return np;
}
```



(20 21)

Exempel:

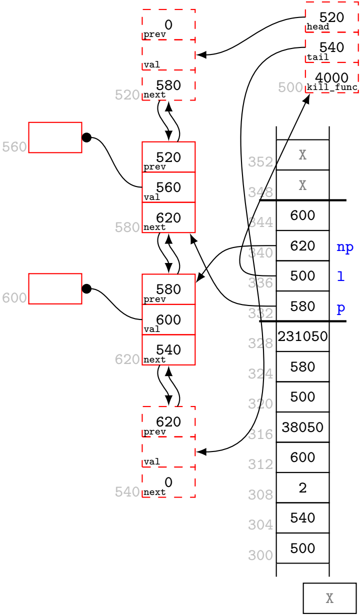
```c
/**
 * list_remove() - Remove an element from a list.
 * @l: List to manipulate.
 * @p: Position in the list of the element to remove.
 *
 * Removes the element at position p from the list. If a l
 * was registered at list creation, calls it to deallocat
 * held by the element value.
 *
 * Returns: The position after the removed element.
 */
list_pos list_remove(list *l, const list_pos p)
{
    // Remember return position.
    list_pos np = p->next;
    // Link past this element.
    p->prev->next = p->next;
    p->next->prev = p->prev;

    // Call kill_func if registered.
    if (l->kill_func != NULL) {
        // Return any user-allocated memory for the value
        l->kill_func(p->val);
    }
    // Return the memory allocated to the cell itself.
    free(p);
    // Return the position of the next element.
    return np;
}
```



(20 21)

Exempel:

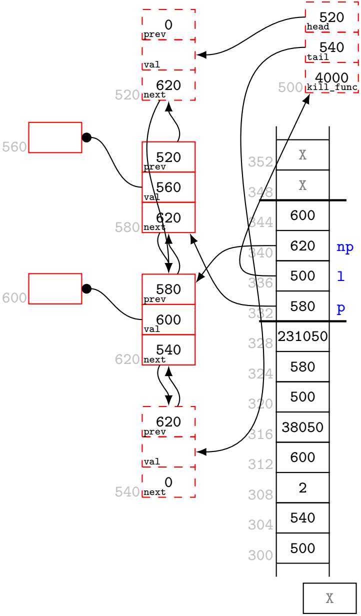
```
/**
 * list_remove() - Remove an element from a list.
 * @l: List to manipulate.
 * @p: Position in the list of the element to remove.
 *
 * Removes the element at position p from the list. If a 
 * was registered at list creation, calls it to deallocat
 * held by the element value.
 *
 * Returns: The position after the removed element.
 */
list_pos list_remove(list *l, const list_pos p)
{
    // Remember return position.
    list_pos np = p->next;
    // Link past this element.
    p->prev->next = p->next;
    p->next->prev = p->prev;

    // Call kill_func if registered.
    if (l->kill_func != NULL) {
        // Return any user-allocated memory for the value
        l->kill_func(p->val);
    }
    // Return the memory allocated to the cell itself.
    free(p);
    // Return the position of the next element.
    return np;
}
```



(20 21)

Exempel:

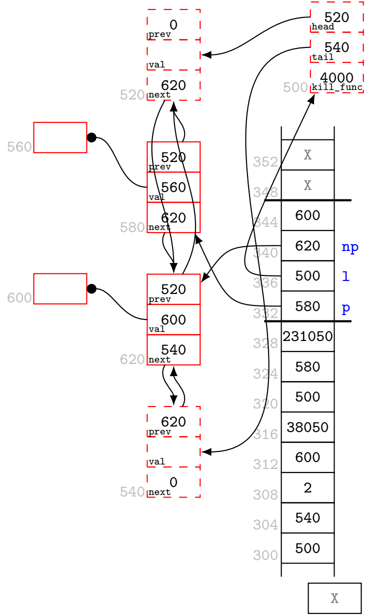
```c
/**
 * list_remove() - Remove an element from a list.
 * @l: List to manipulate.
 * @p: Position in the list of the element to remove.
 *
 * Removes the element at position p from the list. If a
 * was registered at list creation, calls it to deallocat
 * held by the element value.
 *
 * Returns: The position after the removed element.
 */
list_pos list_remove(list *l, const list_pos p)
{
    // Remember return position.
    list_pos np = p->next;
    // Link past this element.
    p->prev->next = p->next;
    p->next->prev = p->prev;

    // Call kill_func if registered.
    if (l->kill_func != NULL) {
        // Return any user-allocated memory for the value
        l->kill_func(p->val);
    }
    // Return the memory allocated to the cell itself.
    free(p);
    // Return the position of the next element.
    return np;
}
```



(20 21)

Exempel:

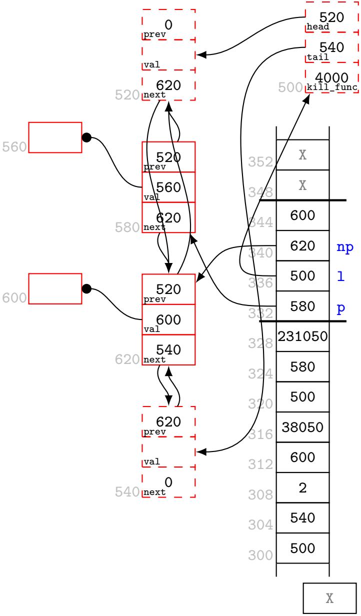
```c
/**
 * list_remove() - Remove an element from a list.
 * @l: List to manipulate.
 * @p: Position in the list of the element to remove.
 *
 * Removes the element at position p from the list. If a
 * was registered at list creation, calls it to deallocat
 * held by the element value.
 *
 * Returns: The position after the removed element.
 */
list_pos list_remove(list *l, const list_pos p)
{
    // Remember return position.
    list_pos np = p->next;
    // Link past this element.
    p->prev->next = p->next;
    p->next->prev = p->prev;

    // Call kill_func if registered.
    if (l->kill_func != NULL) {
        // Return any user-allocated memory for the value
        l->kill_func(p->val);
    }
    // Return the memory allocated to the cell itself.
    free(p);
    // Return the position of the next element.
    return np;
}
```



(20 21)

Exempel:

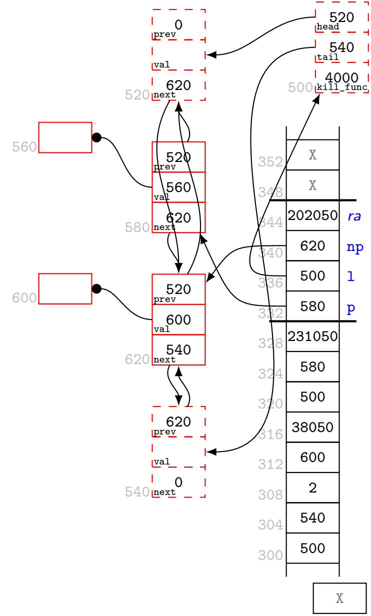
```c
/**
 * list_remove() - Remove an element from a list.
 * @l: List to manipulate.
 * @p: Position in the list of the element to remove.
 *
 * Removes the element at position p from the list. If a l
 * was registered at list creation, calls it to deallocat
 * held by the element value.
 *
 * Returns: The position after the removed element.
 */
list_pos list_remove(list *l, const list_pos p)
{
    // Remember return position.
    list_pos np = p->next;
    // Link past this element.
    p->prev->next = p->next;
    p->next->prev = p->prev;

    // Call kill_func if registered.
    if (l->kill_func != NULL) {
        // Return any user-allocated memory for the value
        l->kill_func(p->val);
    }
    // Return the memory allocated to the cell itself.
    free(p);
    // Return the position of the next element.
    return np;
}
```



(20 21)

Exempel:

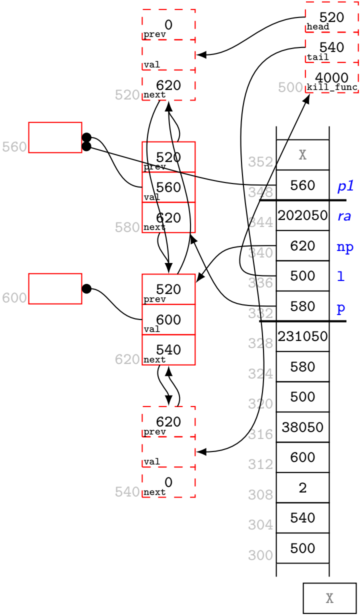
```
/**
 * list_remove() - Remove an element from a list.
 * @l: List to manipulate.
 * @p: Position in the list of the element to remove.
 *
 * Removes the element at position p from the list. If a l
 * was registered at list creation, calls it to deallocat
 * held by the element value.
 *
 * Returns: The position after the removed element.
 */
list_pos list_remove(list *l, const list_pos p)
{
    // Remember return position.
    list_pos np = p->next;
    // Link past this element.
    p->prev->next = p->next;
    p->next->prev = p->prev;

    // Call kill_func if registered.
    if (l->kill_func != NULL) {
        // Return any user-allocated memory for the value
        l->kill_func(p->val);
    }
    // Return the memory allocated to the cell itself.
    free(p);
    // Return the position of the next element.
    return np;
}
```



(20 21)

Exempel:

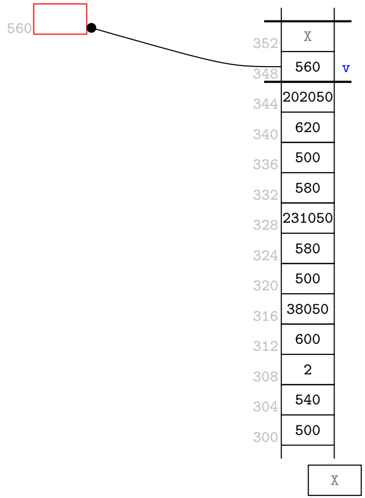
```c
/**
 * list_remove() - Remove an element from a list.
 * @l: List to manipulate.
 * @p: Position in the list of the element to remove.
 *
 * Removes the element at position p from the list. If a 
 * was registered at list creation, calls it to deallocat
 * held by the element value.
 *
 * Returns: The position after the removed element.
 */
list_pos list_remove(list *l, const list_pos p)
{
    // Remember return position.
    list_pos np = p->next;
    // Link past this element.
    p->prev->next = p->next;
    p->next->prev = p->prev;

    // Call kill_func if registered.
    if (l->kill_func != NULL) {
        // Return any user-allocated memory for the value
        l->kill_func(p->val);
    }
    // Return the memory allocated to the cell itself.
    free(p);
    // Return the position of the next element.
    return np;
}
```



(20 21)

Exempel:

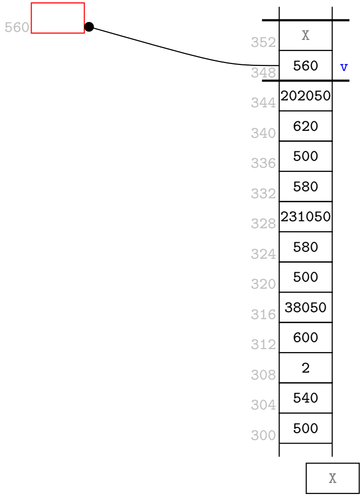
```c
int *int_create(int v)
{
    int *p = malloc(sizeof(*p)); // allocate payload
    *p = v; // assign value
    return p;
}
void int_kill(void *v)
{
    free(v); // deallocate payload
}
int main(void)
{
    // Create empty list, hand over responsibility to
    // deallocate payload using int_kill
    list *l = list_empty(int_kill);
    list_pos p = list_first(l);
    for (int i = 0; i < 2; i++) {
        int *v = int_create(20 + i);
        p = list_insert(l, v, p);
        p = list_next(l, p);
    }
    list_print(l, print_int);

    // Clean up the list, including payload
    list_kill(l);
    return 0;
}
```




```
( 20 21 )
```

Exempel:

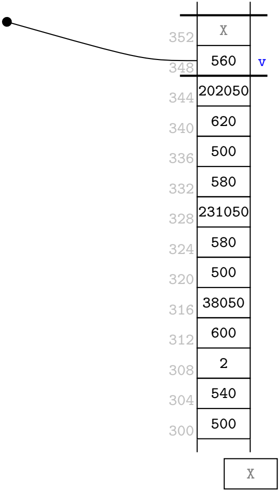
```c
int *int_create(int v)
{
    int *p = malloc(sizeof(*p)); // allocate payload
    *p = v; // assign value
    return p;
}
void int_kill(void *v)
{
    free(v); // deallocate payload
}
int main(void)
{
    // Create empty list, hand over responsibility to
    // deallocate payload using int_kill
    list *l = list_empty(int_kill);
    list_pos p = list_first(l);
    for (int i = 0; i < 2; i++) {
        int *v = int_create(20 + i);
        p = list_insert(l, v, p);
        p = list_next(l, p);
    }
    list_print(l, print_int);





    // Clean up the list, including payload
    list_kill(l);
    return 0;
}
```



(20 21)

Exempel:

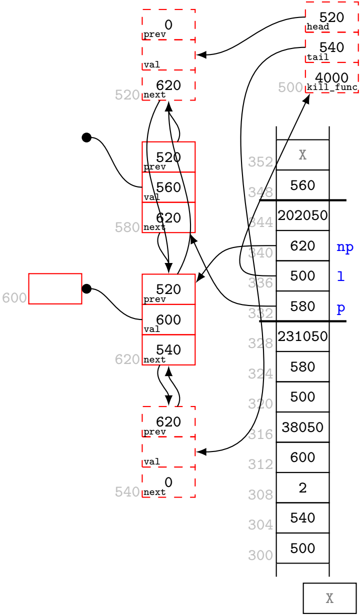
```c
int *int_create(int v)
{
    int *p = malloc(sizeof(*p)); // allocate payload
    *p = v; // assign value
    return p;
}
void int_kill(void *v)
{
    free(v); // deallocate payload
}
int main(void)
{
    // Create empty list, hand over responsibility to
    // deallocate payload using int_kill
    list *l = list_empty(int_kill);
    list_pos p = list_first(l);
    for (int i = 0; i < 2; i++) {
        int *v = int_create(20 + i);
        p = list_insert(l, v, p);
        p = list_next(l, p);
    }
    list_print(l, print_int);






    // Clean up the list, including payload
    list_kill(l);
    return 0;
}
```



(20 21)

Exempel:

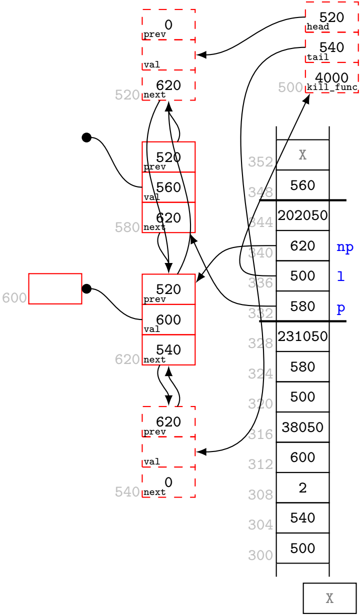
```c
/**
 * list_remove() - Remove an element from a list.
 * @l: List to manipulate.
 * @p: Position in the list of the element to remove.
 *
 * Removes the element at position p from the list. If a 
 * was registered at list creation, calls it to deallocat
 * held by the element value.
 *
 * Returns: The position after the removed element.
 */
list_pos list_remove(list *l, const list_pos p)
{
    // Remember return position.
    list_pos np = p->next;
    // Link past this element.
    p->prev->next = p->next;
    p->next->prev = p->prev;

    // Call kill_func if registered.
    if (l->kill_func != NULL) {
        // Return any user-allocated memory for the value
        l->kill_func(p->val);
    }
    // Return the memory allocated to the cell itself.
    free(p);
    // Return the position of the next element.
    return np;
}
```



(20 21)

Exempel:

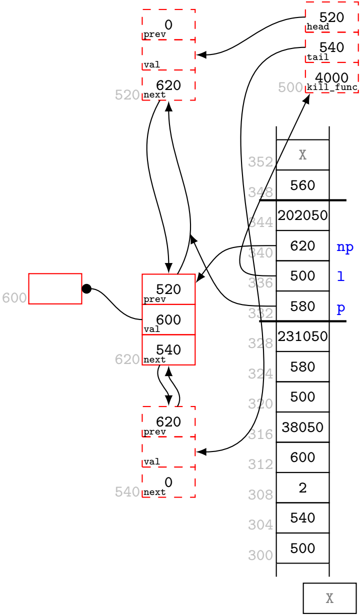
```c
/**
 * list_remove() - Remove an element from a list.
 * @l: List to manipulate.
 * @p: Position in the list of the element to remove.
 *
 * Removes the element at position p from the list. If a 
 * was registered at list creation, calls it to deallocat
 * held by the element value.
 *
 * Returns: The position after the removed element.
 */
list_pos list_remove(list *l, const list_pos p)
{
    // Remember return position.
    list_pos np = p->next;
    // Link past this element.
    p->prev->next = p->next;
    p->next->prev = p->prev;

    // Call kill_func if registered.
    if (l->kill_func != NULL) {
        // Return any user-allocated memory for the value
        l->kill_func(p->val);
    }
    // Return the memory allocated to the cell itself.
    free(p);
    // Return the position of the next element.
    return np;
}
```



(20 21)

Exempel:

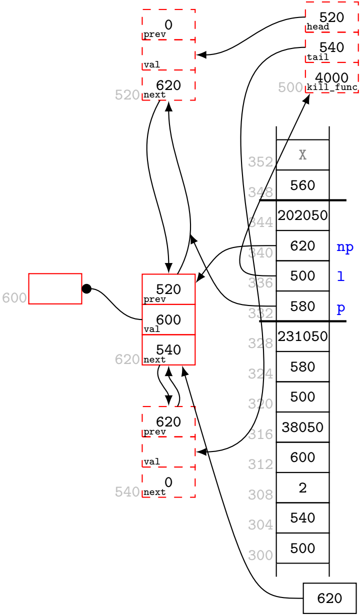
```c
/**
 * list_remove() - Remove an element from a list.
 * @l: List to manipulate.
 * @p: Position in the list of the element to remove.
 *
 * Removes the element at position p from the list. If a 
 * was registered at list creation, calls it to deallocat
 * held by the element value.
 *
 * Returns: The position after the removed element.
 */
list_pos list_remove(list *l, const list_pos p)
{
    // Remember return position.
    list_pos np = p->next;
    // Link past this element.
    p->prev->next = p->next;
    p->next->prev = p->prev;

    // Call kill_func if registered.
    if (l->kill_func != NULL) {
        // Return any user-allocated memory for the value
        l->kill_func(p->val);
    }
    // Return the memory allocated to the cell itself.
    free(p);
    // Return the position of the next element.
    return np;
}
```



(20 21)

Exempel:

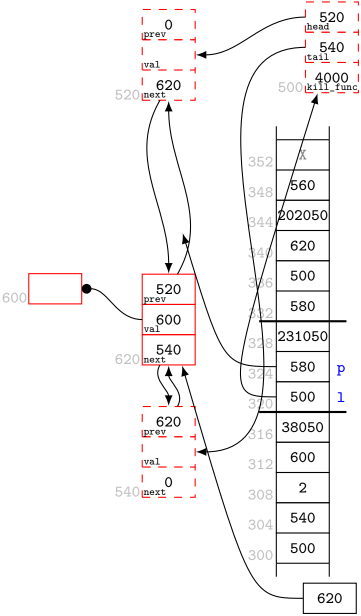
```
/**
 * list_remove() - Remove an element from a list.
 * @l: List to manipulate.
 * @p: Position in the list of the element to remove.
 *
 * Removes the element at position p from the list. If a 
 * was registered at list creation, calls it to deallocat
 * held by the element value.
 *
 * Returns: The position after the removed element.
 */
list_pos list_remove(list *l, const list_pos p)
{
    // Remember return position.
    list_pos np = p->next;
    // Link past this element.
    p->prev->next = p->next;
    p->next->prev = p->prev;

    // Call kill_func if registered.
    if (l->kill_func != NULL) {
        // Return any user-allocated memory for the value
        l->kill_func(p->val);
    }
    // Return the memory allocated to the cell itself.
    free(p);
    // Return the position of the next element.
    return np;
}
```



(20 21)

Exempel:

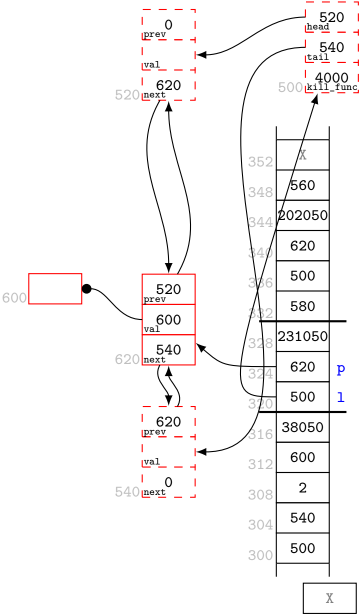
```c
/**
 * list_kill() - Destroy a given list.
 * @l: List to destroy.
 *
 * Returns all dynamic memory used by the list and its el
 * kill_func was registered at list creation, also calls
 * element to return any user-allocated memory occupied b
 * values.
 *
 * Returns: Nothing.
 */
void list_kill(list *l)
{
    // Use public functions to traverse the list.

    // Start with the first element (will be defined
    // even for an empty list).
    list_pos p = list_first(l);

    // Remove first element until list is empty.
    while (!list_is_empty(l)) {
        p = list_remove(l, p);
    }

    // Free border elements and the list head.
    free(l->head);
    free(l->tail);
    free(l);
}
```



(20 21)

Exempel:

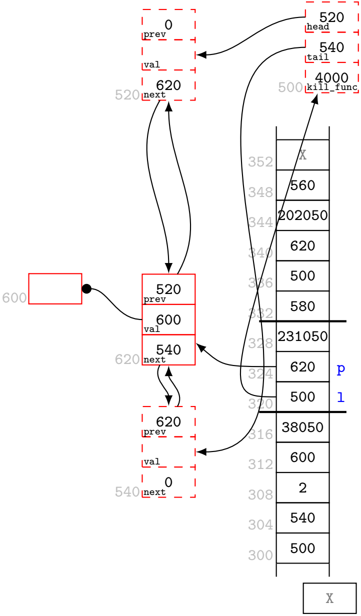
```c
/**
 * list_kill() - Destroy a given list.
 * @l: List to destroy.
 *
 * Returns all dynamic memory used by the list and its el
 * kill_func was registered at list creation, also calls
 * element to return any user-allocated memory occupied b
 * values.
 *
 * Returns: Nothing.
 */
void list_kill(list *l)
{
    // Use public functions to traverse the list.

    // Start with the first element (will be defined
    // even for an empty list).
    list_pos p = list_first(l);

    // Remove first element until list is empty.
    while (!list_is_empty(l)) {
        p = list_remove(l, p);
    }

    // Free border elements and the list head.
    free(l->head);
    free(l->tail);
    free(l);
}
```



(20 21)

Exempel:

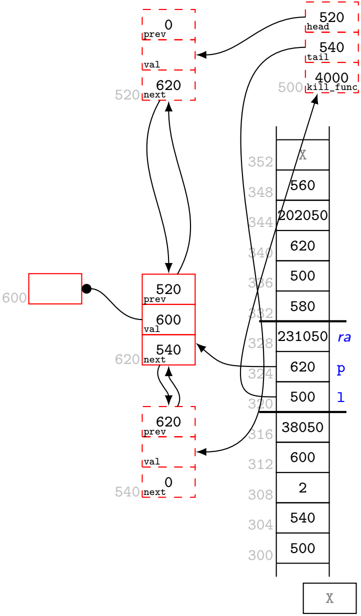
```c
/**
 * list_kill() - Destroy a given list.
 * @l: List to destroy.
 *
 * Returns all dynamic memory used by the list and its el
 * kill_func was registered at list creation, also calls 
 * element to return any user-allocated memory occupied b
 * values.
 *
 * Returns: Nothing.
 */
void list_kill(list *l)
{
    // Use public functions to traverse the list.

    // Start with the first element (will be defined
    // even for an empty list).
    list_pos p = list_first(l);

    // Remove first element until list is empty.
    while (!list_is_empty(l)) {
        p = list_remove(l, p);
    }

    // Free border elements and the list head.
    free(l->head);
    free(l->tail);
    free(l);
}
```



(20 21)

Exempel:

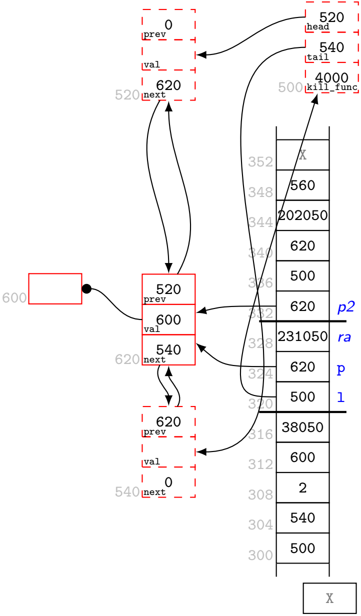
```c
/**
 * list_kill() - Destroy a given list.
 * @l: List to destroy.
 *
 * Returns all dynamic memory used by the list and its el
 * kill_func was registered at list creation, also calls
 * element to return any user-allocated memory occupied b
 * values.
 *
 * Returns: Nothing.
 */
void list_kill(list *l)
{
    // Use public functions to traverse the list.

    // Start with the first element (will be defined
    // even for an empty list).
    list_pos p = list_first(l);

    // Remove first element until list is empty.
    while (!list_is_empty(l)) {
        p = list_remove(l, p);
    }

    // Free border elements and the list head.
    free(l->head);
    free(l->tail);
    free(l);
}
```



(20 21)

Exempel:

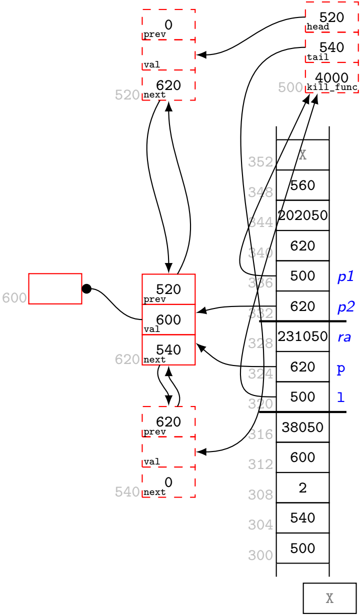
```c
/**
 * list_kill() - Destroy a given list.
 * @l: List to destroy.
 *
 * Returns all dynamic memory used by the list and its el
 * kill_func was registered at list creation, also calls
 * element to return any user-allocated memory occupied b
 * values.
 *
 * Returns: Nothing.
 */
void list_kill(list *l)
{
    // Use public functions to traverse the list.

    // Start with the first element (will be defined
    // even for an empty list).
    list_pos p = list_first(l);

    // Remove first element until list is empty.
    while (!list_is_empty(l)) {
        p = list_remove(l, p);
    }

    // Free border elements and the list head.
    free(l->head);
    free(l->tail);
    free(l);
}
```



(20 21)

Exempel:

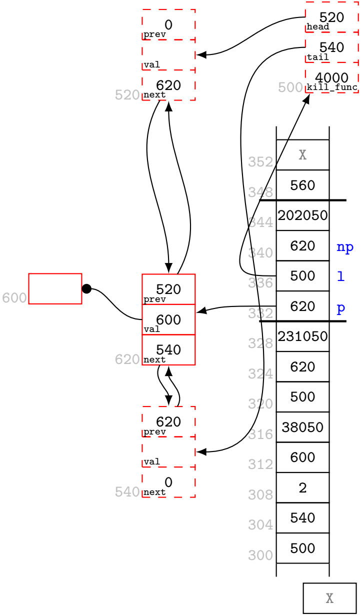
```c
/**
 * list_kill() - Destroy a given list.
 * @l: List to destroy.
 *
 * Returns all dynamic memory used by the list and its el
 * kill_func was registered at list creation, also calls 
 * element to return any user-allocated memory occupied b
 * values.
 *
 * Returns: Nothing.
 */
void list_kill(list *l)
{
    // Use public functions to traverse the list.

    // Start with the first element (will be defined
    // even for an empty list).
    list_pos p = list_first(l);

    // Remove first element until list is empty.
    while (!list_is_empty(l)) {
        p = list_remove(l, p);
    }

    // Free border elements and the list head.
    free(l->head);
    free(l->tail);
    free(l);
}
```



(20 21)

Exempel:

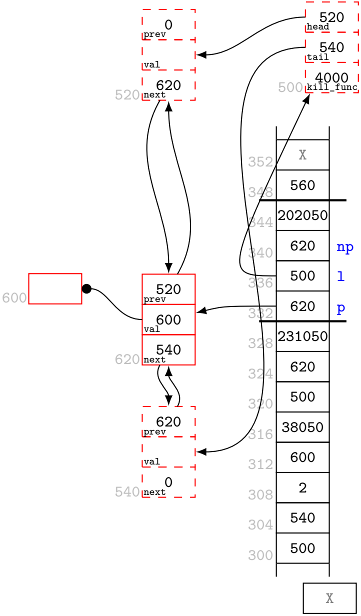
```c
/**
 * list_remove() - Remove an element from a list.
 * @l: List to manipulate.
 * @p: Position in the list of the element to remove.
 *
 * Removes the element at position p from the list. If a l
 * was registered at list creation, calls it to deallocat
 * held by the element value.
 *
 * Returns: The position after the removed element.
 */
list_pos list_remove(list *l, const list_pos p)
{
    // Remember return position.
    list_pos np = p->next;
    // Link past this element.
    p->prev->next = p->next;
    p->next->prev = p->prev;

    // Call kill_func if registered.
    if (l->kill_func != NULL) {
        // Return any user-allocated memory for the value
        l->kill_func(p->val);
    }
    // Return the memory allocated to the cell itself.
    free(p);
    // Return the position of the next element.
    return np;
}
```



(20 21)

Exempel:

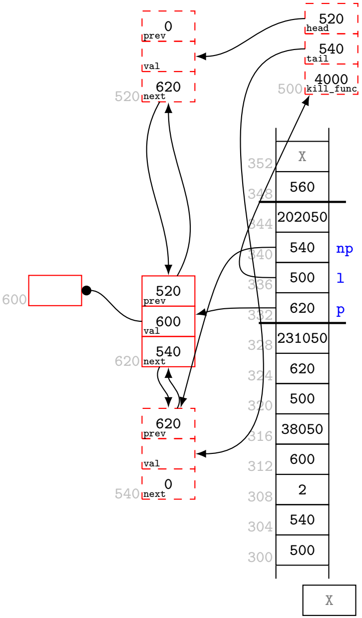
```c
/**
 * list_remove() - Remove an element from a list.
 * @l: List to manipulate.
 * @p: Position in the list of the element to remove.
 *
 * Removes the element at position p from the list. If a 
 * was registered at list creation, calls it to deallocat
 * held by the element value.
 *
 * Returns: The position after the removed element.
 */
list_pos list_remove(list *l, const list_pos p)
{
    // Remember return position.
    list_pos np = p->next;
    // Link past this element.
    p->prev->next = p->next;
    p->next->prev = p->prev;

    // Call kill_func if registered.
    if (l->kill_func != NULL) {
        // Return any user-allocated memory for the value
        l->kill_func(p->val);
    }
    // Return the memory allocated to the cell itself.
    free(p);
    // Return the position of the next element.
    return np;
}
```



(20 21)

Exempel:

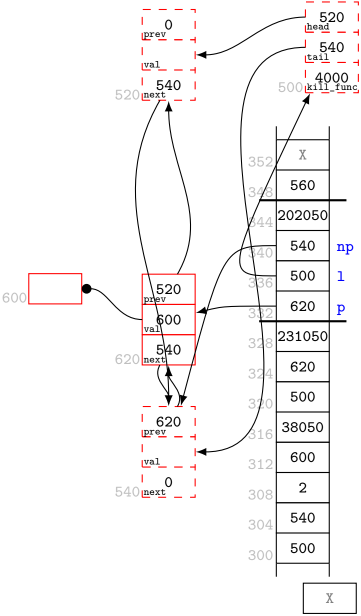
```c
/**
 * list_remove() - Remove an element from a list.
 * @l: List to manipulate.
 * @p: Position in the list of the element to remove.
 *
 * Removes the element at position p from the list. If a
 * was registered at list creation, calls it to deallocat
 * held by the element value.
 *
 * Returns: The position after the removed element.
 */
list_pos list_remove(list *l, const list_pos p)
{
    // Remember return position.
    list_pos np = p->next;
    // Link past this element.
    p->prev->next = p->next;
    p->next->prev = p->prev;

    // Call kill_func if registered.
    if (l->kill_func != NULL) {
        // Return any user-allocated memory for the value
        l->kill_func(p->val);
    }
    // Return the memory allocated to the cell itself.
    free(p);
    // Return the position of the next element.
    return np;
}
```



(20 21)

Exempel:

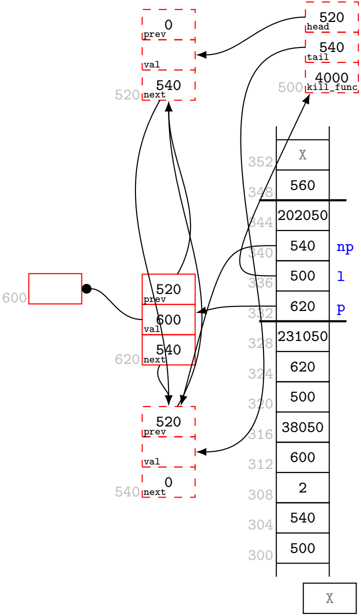
```c
/**
 * list_remove() - Remove an element from a list.
 * @l: List to manipulate.
 * @p: Position in the list of the element to remove.
 *
 * Removes the element at position p from the list. If a 
 * was registered at list creation, calls it to deallocat
 * held by the element value.
 *
 * Returns: The position after the removed element.
 */
list_pos list_remove(list *l, const list_pos p)
{
    // Remember return position.
    list_pos np = p->next;
    // Link past this element.
    p->prev->next = p->next;
    p->next->prev = p->prev;

    // Call kill_func if registered.
    if (l->kill_func != NULL) {
        // Return any user-allocated memory for the value
        l->kill_func(p->val);
    }
    // Return the memory allocated to the cell itself.
    free(p);
    // Return the position of the next element.
    return np;
}
```



(20 21)

Exempel:

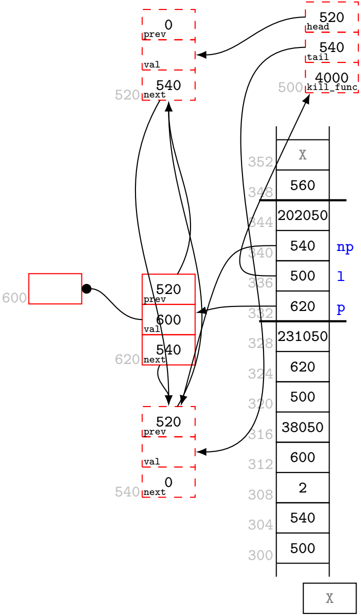
```c
/**
 * list_remove() - Remove an element from a list.
 * @l: List to manipulate.
 * @p: Position in the list of the element to remove.
 *
 * Removes the element at position p from the list. If a 
 * was registered at list creation, calls it to deallocat
 * held by the element value.
 *
 * Returns: The position after the removed element.
 */
list_pos list_remove(list *l, const list_pos p)
{
    // Remember return position.
    list_pos np = p->next;
    // Link past this element.
    p->prev->next = p->next;
    p->next->prev = p->prev;

    // Call kill_func if registered.
    if (l->kill_func != NULL) {
        // Return any user-allocated memory for the value
        l->kill_func(p->val);
    }
    // Return the memory allocated to the cell itself.
    free(p);
    // Return the position of the next element.
    return np;
}
```



(20 21)

Exempel:

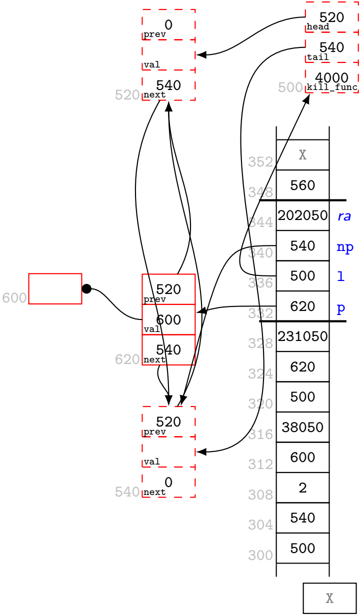
```c
/**
 * list_remove() - Remove an element from a list.
 * @l: List to manipulate.
 * @p: Position in the list of the element to remove.
 *
 * Removes the element at position p from the list. If a l
 * was registered at list creation, calls it to deallocat
 * held by the element value.
 *
 * Returns: The position after the removed element.
 */
list_pos list_remove(list *l, const list_pos p)
{
    // Remember return position.
    list_pos np = p->next;
    // Link past this element.
    p->prev->next = p->next;
    p->next->prev = p->prev;

    // Call kill_func if registered.
    if (l->kill_func != NULL) {
        // Return any user-allocated memory for the value
        l->kill_func(p->val);
    }
    // Return the memory allocated to the cell itself.
    free(p);
    // Return the position of the next element.
    return np;
}
```



(20 21)

Exempel:

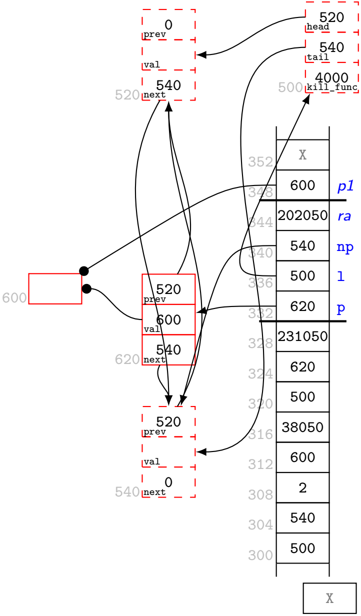
```c
/**
 * list_remove() - Remove an element from a list.
 * @l: List to manipulate.
 * @p: Position in the list of the element to remove.
 *
 * Removes the element at position p from the list. If a 
 * was registered at list creation, calls it to deallocat
 * held by the element value.
 *
 * Returns: The position after the removed element.
 */
list_pos list_remove(list *l, const list_pos p)
{
    // Remember return position.
    list_pos np = p->next;
    // Link past this element.
    p->prev->next = p->next;
    p->next->prev = p->prev;

    // Call kill_func if registered.
    if (l->kill_func != NULL) {
        // Return any user-allocated memory for the value
        l->kill_func(p->val);
    }
    // Return the memory allocated to the cell itself.
    free(p);
    // Return the position of the next element.
    return np;
}
```



(20 21)

Exempel:

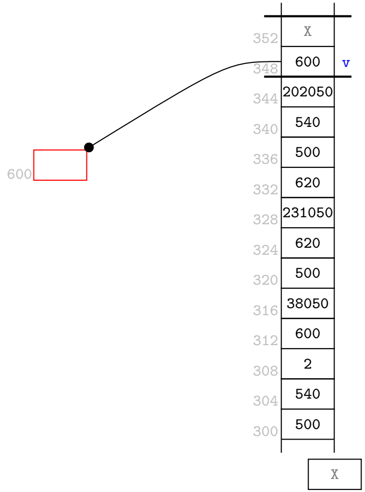
```
/**
 * list_remove() - Remove an element from a list.
 * @l: List to manipulate.
 * @p: Position in the list of the element to remove.
 *
 * Removes the element at position p from the list. If a k
 * was registered at list creation, calls it to deallocat
 * held by the element value.
 *
 * Returns: The position after the removed element.
 */
list_pos list_remove(list *l, const list_pos p)
{
    // Remember return position.
    list_pos np = p->next;
    // Link past this element.
    p->prev->next = p->next;
    p->next->prev = p->prev;

    // Call kill_func if registered.
    if (l->kill_func != NULL) {
        // Return any user-allocated memory for the value
        l->kill_func(p->val);
    }
    // Return the memory allocated to the cell itself.
    free(p);
    // Return the position of the next element.
    return np;
}
```



(20 21)

Exempel:

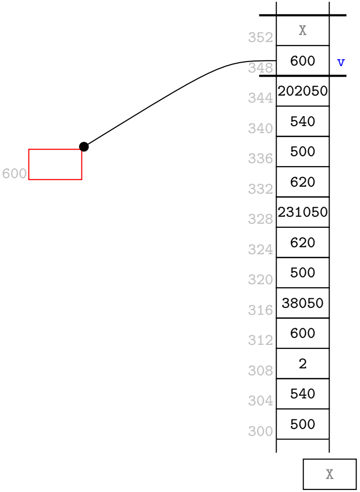
```c
int *int_create(int v)
{
    int *p = malloc(sizeof(*p)); // allocate payload
    *p = v; // assign value
    return p;
}
void int_kill(void *v)
{
    free(v); // deallocate payload
}
int main(void)
{
    // Create empty list, hand over responsibility to
    // deallocate payload using int_kill
    list *l = list_empty(int_kill);
    list_pos p = list_first(l);
    for (int i = 0; i < 2; i++) {
        int *v = int_create(20 + i);
        p = list_insert(l, v, p);
        p = list_next(l, p);
    }
    list_print(l, print_int);
    // Clean up the list, including payload
    list_kill(l);
    return 0;
}
```



(20 21)

Exempel:

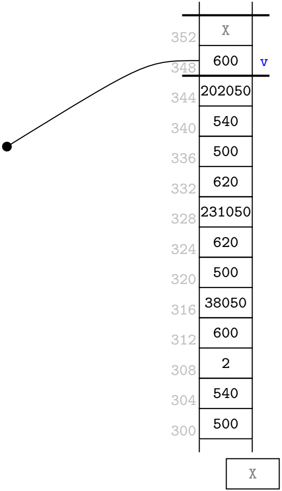
```
int *int_create(int v)
{
    int *p = malloc(sizeof(*p)); // allocate payload
    *p = v; // assign value
    return p;
}
void int_kill(void *v)
{
    free(v); // deallocate payload
}
int main(void)
{
    // Create empty list, hand over responsibility to
    // deallocate payload using int_kill
    list *l = list_empty(int_kill);
    list_pos p = list_first(l);
    for (int i = 0; i < 2; i++) {
        int *v = int_create(20 + i);
        p = list_insert(l, v, p);
        p = list_next(l, p);
    }
    list_print(l, print_int);






    // Clean up the list, including payload
    list_kill(l);
    return 0;
}
```



(20 21)

Exempel:

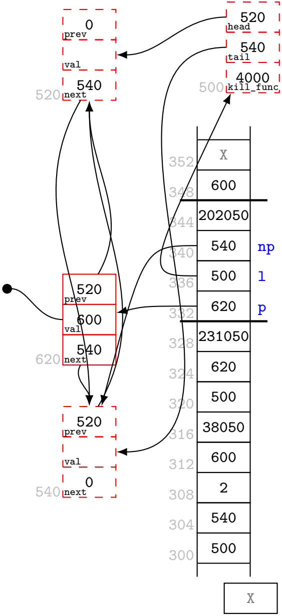
```c
int *int_create(int v)
{
    int *p = malloc(sizeof(*p)); // allocate payload
    *p = v; // assign value
    return p;
}
void int_kill(void *v)
{
    free(v); // deallocate payload
}
int main(void)
{
    // Create empty list, hand over responsibility to
    // deallocate payload using int_kill
    list *l = list_empty(int_kill);
    list_pos p = list_first(l);
    for (int i = 0; i < 2; i++) {
        int *v = int_create(20 + i);
        p = list_insert(l, v, p);
        p = list_next(l, p);
    }
    list_print(l, print_int);




    // Clean up the list, including payload
    list_kill(l);
    return 0;
}
```



(20 21)

Exempel:

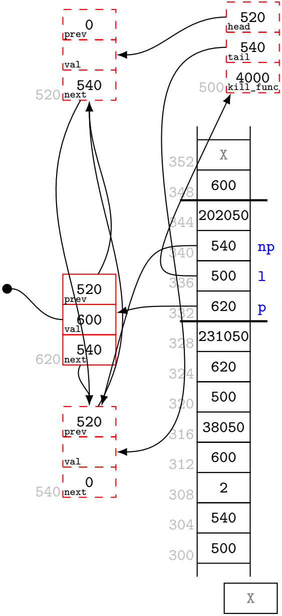
```c
/**
 * list_remove() - Remove an element from a list.
 * @l: List to manipulate.
 * @p: Position in the list of the element to remove.
 *
 * Removes the element at position p from the list. If a 
 * was registered at list creation, calls it to deallocat
 * held by the element value.
 *
 * Returns: The position after the removed element.
 */
list_pos list_remove(list *l, const list_pos p)
{
    // Remember return position.
    list_pos np = p->next;
    // Link past this element.
    p->prev->next = p->next;
    p->next->prev = p->prev;

    // Call kill_func if registered.
    if (l->kill_func != NULL) {
        // Return any user-allocated memory for the value
        l->kill_func(p->val);
    }
    // Return the memory allocated to the cell itself.
    free(p);
    // Return the position of the next element.
    return np;
}
```



(20 21)

Exempel:

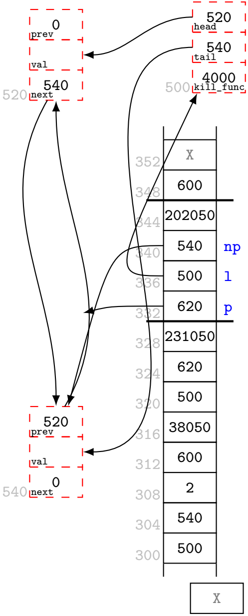
```c
/**
 * list_remove() - Remove an element from a list.
 * @l: List to manipulate.
 * @p: Position in the list of the element to remove.
 *
 * Removes the element at position p from the list. If a k
 * was registered at list creation, calls it to deallocat
 * held by the element value.
 *
 * Returns: The position after the removed element.
 */
list_pos list_remove(list *l, const list_pos p)
{
    // Remember return position.
    list_pos np = p->next;
    // Link past this element.
    p->prev->next = p->next;
    p->next->prev = p->prev;

    // Call kill_func if registered.
    if (l->kill_func != NULL) {
        // Return any user-allocated memory for the value
        l->kill_func(p->val);
    }
    // Return the memory allocated to the cell itself.
    free(p);
    // Return the position of the next element.
    return np;
}
```



(20 21)

Exempel:

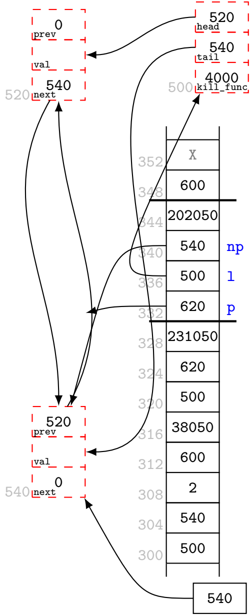
```c
/**
 * list_remove() - Remove an element from a list.
 * @l: List to manipulate.
 * @p: Position in the list of the element to remove.
 *
 * Removes the element at position p from the list. If a 
 * was registered at list creation, calls it to deallocat
 * held by the element value.
 *
 * Returns: The position after the removed element.
 */
list_pos list_remove(list *l, const list_pos p)
{
    // Remember return position.
    list_pos np = p->next;
    // Link past this element.
    p->prev->next = p->next;
    p->next->prev = p->prev;

    // Call kill_func if registered.
    if (l->kill_func != NULL) {
        // Return any user-allocated memory for the value
        l->kill_func(p->val);
    }
    // Return the memory allocated to the cell itself.
    free(p);
    // Return the position of the next element.
    return np;
}
```



(20 21)

Exempel:

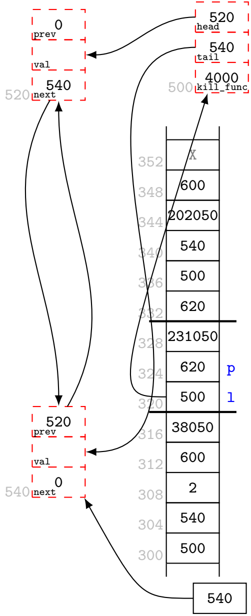
```c
/**
 * list_remove() - Remove an element from a list.
 * @l: List to manipulate.
 * @p: Position in the list of the element to remove.
 *
 * Removes the element at position p from the list. If a
 * was registered at list creation, calls it to deallocat
 * held by the element value.
 *
 * Returns: The position after the removed element.
 */
list_pos list_remove(list *l, const list_pos p)
{
    // Remember return position.
    list_pos np = p->next;
    // Link past this element.
    p->prev->next = p->next;
    p->next->prev = p->prev;

    // Call kill_func if registered.
    if (l->kill_func != NULL) {
        // Return any user-allocated memory for the value
        l->kill_func(p->val);
    }
    // Return the memory allocated to the cell itself.
    free(p);
    // Return the position of the next element.
    return np;
}
```



(20 21)

Exempel:

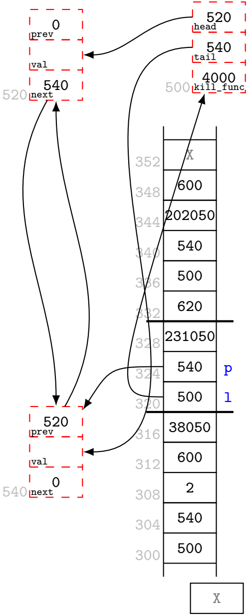
```c
/**
 * list_kill() - Destroy a given list.
 * @l: List to destroy.
 *
 * Returns all dynamic memory used by the list and its el
 * kill_func was registered at list creation, also calls 
 * element to return any user-allocated memory occupied b
 * values.
 *
 * Returns: Nothing.
 */
void list_kill(list *l)
{
    // Use public functions to traverse the list.

    // Start with the first element (will be defined
    // even for an empty list).
    list_pos p = list_first(l);

    // Remove first element until list is empty.
    while (!list_is_empty(l)) {
        p = list_remove(l, p);
    }

    // Free border elements and the list head.
    free(l->head);
    free(l->tail);
    free(l);
}
```



(20 21)

Exempel:

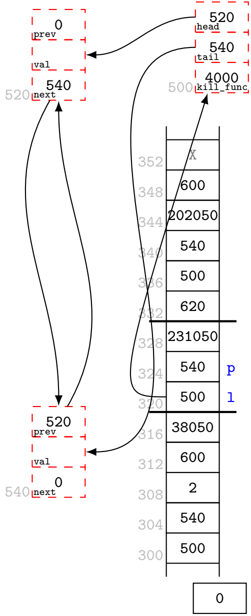
```
/**
 * list_kill() - Destroy a given list.
 * @l: List to destroy.
 *
 * Returns all dynamic memory used by the list and its el
 * kill_func was registered at list creation, also calls
 * element to return any user-allocated memory occupied b
 * values.
 *
 * Returns: Nothing.
 */
void list_kill(list *l)
{
    // Use public functions to traverse the list.

    // Start with the first element (will be defined
    // even for an empty list).
    list_pos p = list_first(l);

    // Remove first element until list is empty.
    while (!list_is_empty(l)) {
        p = list_remove(l, p);
    }

    // Free border elements and the list head.
    free(l->head);
    free(l->tail);
    free(l);
}
```



(20 21)

Exempel:

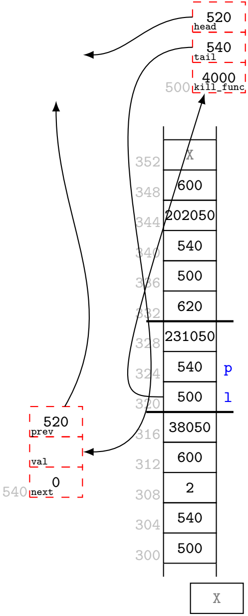
```
/**
 * list_kill() - Destroy a given list.
 * @l: List to destroy.
 *
 * Returns all dynamic memory used by the list and its el
 * kill_func was registered at list creation, also calls
 * element to return any user-allocated memory occupied b
 * values.
 *
 * Returns: Nothing.
 */
void list_kill(list *l)
{
    // Use public functions to traverse the list.

    // Start with the first element (will be defined
    // even for an empty list).
    list_pos p = list_first(l);

    // Remove first element until list is empty.
    while (!list_is_empty(l)) {
        p = list_remove(l, p);
    }

    // Free border elements and the list head.
    free(l->head);
    free(l->tail);
    free(l);
}
```



(20 21)

Exempel:

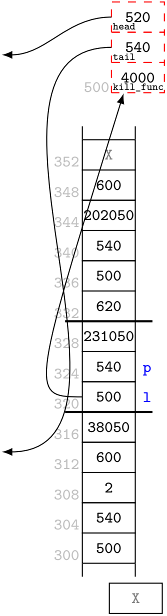
```c
/**
 * list_kill() - Destroy a given list.
 * @l: List to destroy.
 *
 * Returns all dynamic memory used by the list and its el
 * kill_func was registered at list creation, also calls
 * element to return any user-allocated memory occupied b
 * values.
 *
 * Returns: Nothing.
 */
void list_kill(list *l)
{
    // Use public functions to traverse the list.

    // Start with the first element (will be defined
    // even for an empty list).
    list_pos p = list_first(l);

    // Remove first element until list is empty.
    while (!list_is_empty(l)) {
        p = list_remove(l, p);
    }

    // Free border elements and the list head.
    free(l->head);
    free(l->tail);
    free(l);
}
```



(20 21)

Exempel:

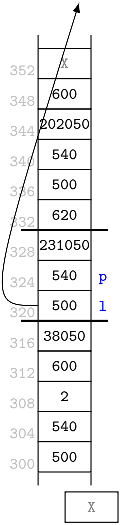
```c
/**
 * list_kill() - Destroy a given list.
 * @l: List to destroy.
 *
 * Returns all dynamic memory used by the list and its el
 * kill_func was registered at list creation, also calls 
 * element to return any user-allocated memory occupied b
 * values.
 *
 * Returns: Nothing.
 */
void list_kill(list *l)
{
    // Use public functions to traverse the list.

    // Start with the first element (will be defined
    // even for an empty list).
    list_pos p = list_first(l);

    // Remove first element until list is empty.
    while (!list_is_empty(l)) {
        p = list_remove(l, p);
    }

    // Free border elements and the list head.
    free(l->head);
    free(l->tail);
    free(l);
}
```



(20 21)

Exempel:

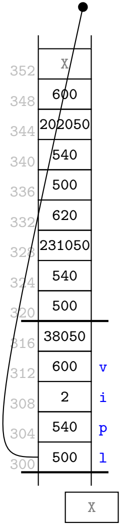
```c
/**
 * list_kill() - Destroy a given list.
 * @l: List to destroy.
 *
 * Returns all dynamic memory used by the list and its el
 * kill_func was registered at list creation, also calls
 * element to return any user-allocated memory occupied b
 * values.
 *
 * Returns: Nothing.
 */
void list_kill(list *l)
{
    // Use public functions to traverse the list.

    // Start with the first element (will be defined
    // even for an empty list).
    list_pos p = list_first(l);

    // Remove first element until list is empty.
    while (!list_is_empty(l)) {
        p = list_remove(l, p);
    }

    // Free border elements and the list head.
    free(l->head);
    free(l->tail);
    free(l);
}
```



(20 21)

Exempel:

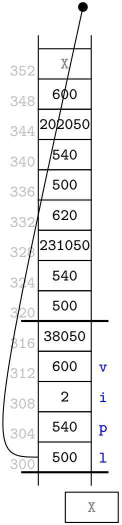
```c
int *int_create(int v)
{
    int *p = malloc(sizeof(*p)); // allocate payload
    *p = v; // assign value
    return p;
}
void int_kill(void *v)
{
    free(v); // deallocate payload
}
int main(void)
{
    // Create empty list, hand over responsibility to
    // deallocate payload using int_kill
    list *l = list_empty(int_kill);
    list_pos p = list_first(l);
    for (int i = 0; i < 2; i++) {
        int *v = int_create(20 + i);
        p = list_insert(l, v, p);
        p = list_next(l, p);
    }
    list_print(l, print_int);

    // Clean up the list, including payload
    list_kill(l);
    return 0;
}
```



(20 21)

Exempel:

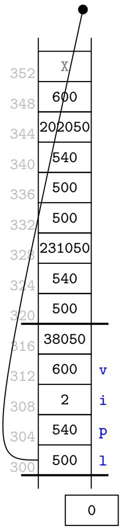
```c
int *int_create(int v)
{
    int *p = malloc(sizeof(*p)); // allocate payload
    *p = v; // assign value
    return p;
}
void int_kill(void *v)
{
    free(v); // deallocate payload
}
int main(void)
{
    // Create empty list, hand over responsibility to
    // deallocate payload using int_kill
    list *l = list_empty(int_kill);
    list_pos p = list_first(l);
    for (int i = 0; i < 2; i++) {
        int *v = int_create(20 + i);
        p = list_insert(l, v, p);
        p = list_next(l, p);
    }
    list_print(l, print_int);





    // Clean up the list, including payload
    list_kill(l);
    return 0;
}
```



(20 21)

Exempel:

```c
int *int_create(int v)
{
    int *p = malloc(sizeof(*p)); // allocate payload
    *p = v; // assign value
    return p;
}
void int_kill(void *v)
{
    free(v); // deallocate payload
}
int main(void)
{
    // Create empty list, hand over responsibility to
    // deallocate payload using int_kill
    list *l = list_empty(int_kill);
    list_pos p = list_first(l);
    for (int i = 0; i < 2; i++) {
        int *v = int_create(20 + i);
        p = list_insert(l, v, p);
        p = list_next(l, p);
    }
    list_print(l, print_int);






    // Clean up the list, including payload
    list_kill(l);
    return 0;
}
```



(20 21)

Exempel:

```c
int *int_create(int v)
{
    int *p = malloc(sizeof(*p)); // allocate payload
    *p = v; // assign value
    return p;
}
void int_kill(void *v)
{
    free(v); // deallocate payload
}
int main(void)
{
    // Create empty list, hand over responsibility to
    // deallocate payload using int_kill
    list *l = list_empty(int_kill);
    list_pos p = list_first(l);
    for (int i = 0; i < 2; i++) {
        int *v = int_create(20 + i);
        p = list_insert(l, v, p);
        p = list_next(l, p);
    }
    list_print(l, print_int);






    // Clean up the list, including payload
    list_kill(l);
    return 0;
}
```

352	X
348	600
344	202050
340	540
336	500
332	500
328	231050
324	540
320	500
316	38050
312	600
308	2
304	540
300	500

0

(20 21)

Ansvarsförvirring

Ansvarsförvirring

- Ansvaret för att lämna tillbaka minne som allokerats dynamiskt till listans komponenter (huvud, 2-celler) ligger hos listans funktioner
- Vem har ansvar för att lämna tillbaka minne som allokerats dynamiskt till payload?
- Här finns två alternativ:
  1. Ansvaret för att återlämna minnet behålls av den del av programmet som reserverade minnet
     - Vi kan kalla det för att listan lånat minnet
     - Återlämningen sker i kod som ligger utanför listan
  2. Listan övertar ansvaret för att återlämna minnet
     - Vi kan kalla det för att listan äger minnet
     - Återlämningen sker i kod som tillhör listan

The `kill_function` (1)

- I kodbasen hanteras detta genom att varje generisk datatyp, t.ex. Stack, Lista, etc., har en s.k. `kill_function` kopplad till sig
  - Tilldelningen av en `kill_function` sker vid konstruktionen av datatypen och är kopplat till den instansen av datatypen
  - Det betyder att vissa instanser kan överta ansvaret medan andra gör det inte

The `kill_function` (2)

- Anropet

```
list *l = list_empty(int_kill);
```

  betyder att när ett element tas bort från listan så anropas funktionen `int_kill` med den lagrade pekaren
  - Det motsvarar att listan äger minnet
- Anropet

```
list *l = list_empty(NULL);
```

  betyder att när ett element tas bort från listan så anropas ingen funktion med den lagrade pekaren
  - Det motsvarar att listan lånat minnet

När väljer man vad? (1)

- När väljer man att använda en `kill_function`?

```
list *l = list_empty(int_kill);
```

  - Enklare kod
  - Aktuellt för OU1
  - Lagra en egen kopia av varje värde

När väljer man vad? (2)

- När väljer man att inte använda en `kill_function`?

```
list *l = list_empty(NULL);
```

  - Större kontroll över när elementvärden avallokeras
    - Aktuellt för OU3
  - Om elementvärdena inte är dynamiskt allokerade
  - Gör det möjligt att lagra referenser till värden utan att kopiera dem

När väljer man vad? (3)

- Om vi vill lagra samma värden i flera listor (eller i någon annan datatyp) kan vi kombinera alternativen:
  - En lista äger datatat (har en `kill_function`)
  - En eller flera andra listor lånar minnet (saknar `kill_function`)
    - Troligen aktuellt för OU4
- OBS! Maximalt en datatyp kan äga ett elementvärde!

Jämförelse med och utan `kill_function`

- Lista 3a använder en `kill_function`

code/list-with-freehandler.c

```c
int main(void)
{
    // Create empty list, hand over responsibility to
    // deallocate payload using int_kill
    list *l = list_empty(int_kill);
    list_pos p = list_first(l);
    for (int i = 0; i < 2; i++) {
        int *v = int_create(20 + i);
        p = list_insert(l, v, p);
        p = list_next(l, p);
    }
    list_print(l, print_int);






    // Clean up the list, including payload
    list_kill(l);
    return 0;
}
```

- Motsvarar grovt `list_mwe2.c`

- Lista 3b använder ingen `kill_function`

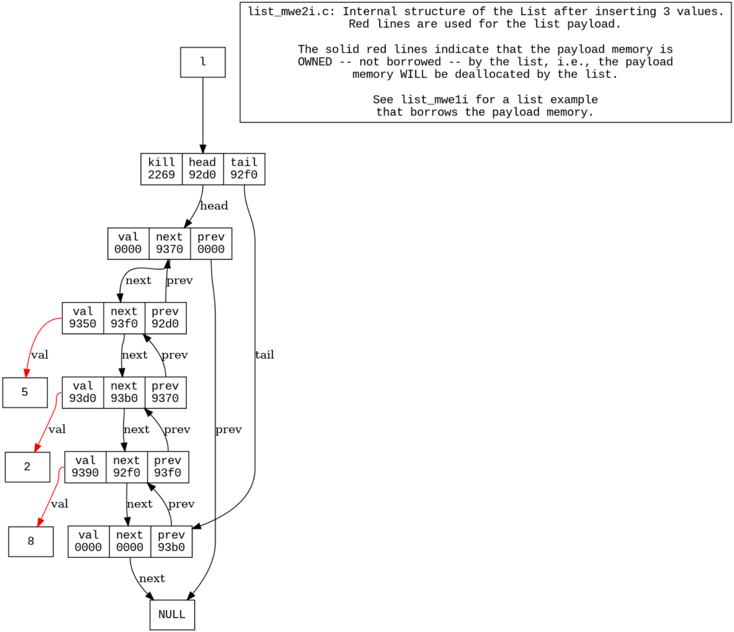
code/list-no-freehandler.c

```c
int main(void)
{
    // Create empty list, keep responsibility to
    // deallocate payload
    list *l = list_empty(NULL);
    list_pos p = list_first(l);
    for (int i = 0; i < 2; i++) {
        int *v = int_create(20 + i);
        p = list_insert(l, v, p);
        p = list_next(l, p);
    }
    list_print(l, print_int);
    // Clean up the payloads
    p = list_first(l);
    while (!list_pos_is_equal(l, p, list_end(l))) {
        int *v = list_inspect(l, p);
        int_kill(v); // free payload
        p = list_next(l, p);
    }
    list_kill(l); // Clean up the list
    return 0;
}
```

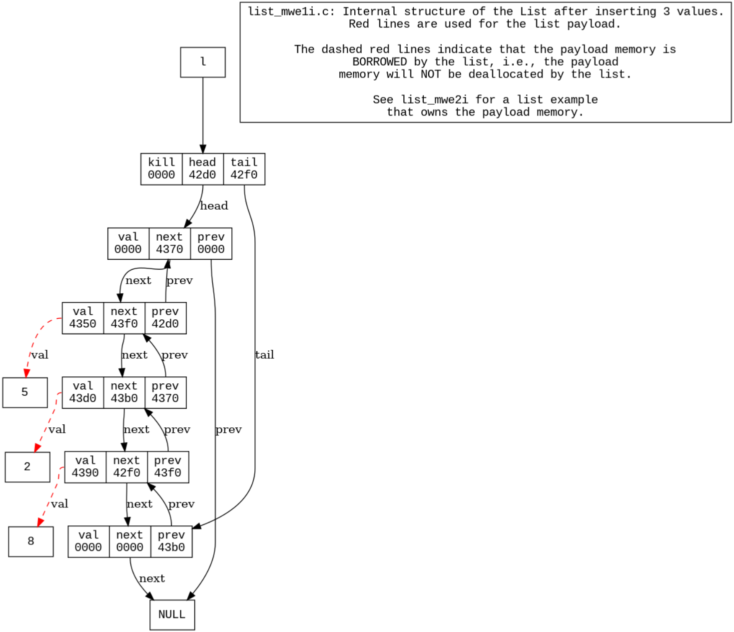
- Motsvarar grovt `list_mwe1.c`

Blank

list_mwe2i.c


list_mwe2i.c: Internal structure of the List after inserting 3 values.
Red lines are used for the list payload.
The solid red lines indicate that the payload memory is
OWNED -- not borrowed -- by the list, i.e., the payload
memory WILL be deallocated by the list.
See list_mwe1i for a list example
that borrows the payload memory.
l
kill 2269
head 92d0
tail 92f0
head
tail
val 0000
next 9370
prev 0000
next
prev
val 9350
next 93f0
prev 92d0
val
next
prev
5
val 93d0
next 93b0
prev 9370
val
next
prev
2
val 9390
next 92f0
prev 93f0
val
next
prev
8
val 0000
next 0000
prev 93b0
next
NULL

list_mwe1i.c


list_mwe1i.c: Internal structure of the List after inserting 3 values.
Red lines are used for the list payload.
The dashed red lines indicate that the payload memory is
BORROWED by the list, i.e., the payload
memory will NOT be deallocated by the list.
See list_mwe2i for a list example
that owns the payload memory.
l
kill 0000
head 42d0
tail 42f0
head
val 0000
next 4370
prev 0000
next
prev
val 4350
next 43f0
prev 42d0
val
next
prev
tail
5
val 43d0
next 43b0
prev 4370
val
next
prev
prev
2
val 4390
next 42f0
prev 43f0
val
next
prev
8
val 0000
next 0000
prev 43b0
next
NULL