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
     #include <stdio.h>
1
2
     void print(int v)
3
     {
         printf("v = %d\n",v);
4
5
     int add(int c, int d)
6
7
8
         int a:
         a = c + d;
9
         return a:
10
11
     int main(void)
12
13
         int a = 2;
14
         int c = 3:
15
16
         int sum:
         sum = add(a, c);
17
         print(sum);
18
         sum = add(sum, c + 4);
19
20
         print(sum);
         return 0;
21
22
     }
```

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;
```

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 lagras i en minnesarea som kallas stacken

```
code/vars-on-stack.c

int main(void)

{

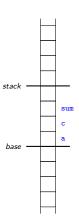
int a = 2;

int c = 3;

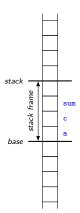
int sum;
```

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

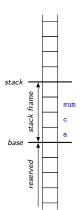
► Internt används två pekare (base och stack) för att hålla reda på funktionens del av stacken



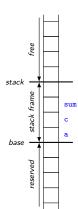
- ► Internt används två pekare (base och stack) för att hålla reda på funktionens del av stacken
 - Området mellan pekarna kallas för stack frame (aktiveringspost)



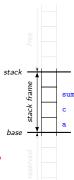
- ► Internt används två pekare (base och stack) för att hålla reda på funktionens del av stacken
 - Området mellan pekarna kallas för stack frame (aktiveringspost)
 - Området under kan betraktas som upptaget



- ► 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



- ► 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)



12

13

15

16

```
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



```
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

12

13

- ▶ 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)



```
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

12 13

- ▶ 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



```
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

12

13

- ▶ 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)



- 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

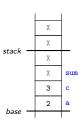
men inte

- variabelns adress
- variabelns storlek

```
code/vars-on-stack.c

int main(void)

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



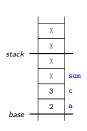
 Stacken används också för parameteröverföring vid funktionsanrop

```
code/vars-on-stack.c

int main(void)

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

Koden på rad 17 översätts av kompilatorn till ungefär följande operationer



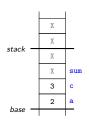
- 1) Beräkna och lagra en returadress på stacken
- 2) Evaluera uttrycket för parameter 2 och lägg det på stacken
- 3) Evaluera uttrycket för parameter 1 och lägg det på stacken
- 4) Hoppa till funktionen add
- 5) Ta hand om returvärdet

```
code/vars-on-stack.c

int main(void)

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

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



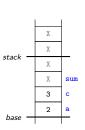
- 1) Beräkna och lagra en returadress på stacken
- 2) Evaluera uttrycket för parameter 2 och lägg det på stacken
- 3) Evaluera uttrycket för parameter 1 och lägg det på stacken
- 4) Hoppa till funktionen add
- 5) Ta hand om returvärdet

```
code/vars-on-stack.c

int main(void)

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

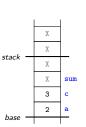
- Koden på rad 17 översätts av kompilatorn till ungefär följande operationer
- Den översatta maskinkoden kommer att hamna någonstans i minnet
 - Jag illustrerar med fejkade adresser som börjar på radnumret
 - ▶ Varje instruktion antas ta 10 bytes
 - 1) Beräkna och lagra en returadress på stacken
 - 2) Evaluera uttrycket för parameter 2 och lägg det på stacken
 - 3) Evaluera uttrycket för parameter 1 och lägg det på stacken
 - Hoppa till funktionen add
 - 5) Ta hand om returvärdet



```
code/vars-on-stack.c

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



 Stacken används också för parameteröverföring vid funktionsanrop

```
code/vars-on-stack.c

int main(void)

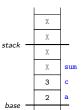
{
    int a = 2;
    int c = 3;
    int sum;
    sum = add(a, c);
```



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



 Stacken används också för parameteröverföring vid funktionsanrop

```
code/vars-on-stack.c
      int main(void)
12
      {
13
14
          int a = 2;
15
          int c = 3:
16
          int sum:
17
          sum = add(a, c);
```

I mer detalj så översätts instruktionerna

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

```
store 17050 at (stack-4)
                         // ret addr
```

17020 17030

X

χ

3

2

return address

sum

stack

 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
         Evaluera uttrycket för parameter 2 och lägg det på stacken
17020
         Evaluera uttrycket för parameter 1 och lägg det på stacken
17030
         Hoppa till funktionen add
17040
17050
         Ta hand om returvärdet
till
17010
        store 17050 at (stack-4) // ret addr
17020
17030
17040
17050
```

hase

 Stacken används också för parameteröverföring vid funktionsanrop

```
code/vars-on-stack.c

int main(void)

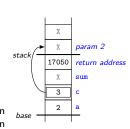
{
    int a = 2;
    int c = 3;
    int sum;

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 store 17050 at (stack-4) // ret addr
```

store (base+4) at (stack+0) // c -> p2



17020

 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

store 17050 at (stack-4) // ret addr

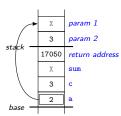
store (base+4) at (stack+0) // c -> p2

 Stacken används också för parameteröverföring vid funktionsanrop

```
code/vars-on-stack.c
     int main(void)
12
     {
13
14
          int a = 2;
15
          int c = 3;
16
          int sum:
          sum = add(a, c);
17
```

I mer detalj så översätts instruktionerna

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



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

 Stacken används också för parameteröverföring vid funktionsanrop

```
code/vars-on-stack.c
     int main(void)
12
     {
13
          int a = 2;
14
15
          int c = 3;
16
          int sum:
          sum = add(a, c);
17
```

I mer detalj så översätts instruktionerna

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

```
param 1
                param 2
stack
         17050
                return address
           Χ
                sum
           3
           2
hase
```

```
till
```

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

 Stacken används också för parameteröverföring vid funktionsanrop

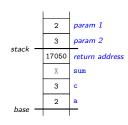
```
code/vars-on-stack.c
     int main(void)
12
     {
13
          int a = 2;
14
15
          int c = 3;
16
          int sum:
17
          sum = |add(a, c);
```

I mer detalj så översätts instruktionerna

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

```
param 1
                param 2
stack
         17050
                return address
           Χ
                sum
           3
           2
hase
```

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



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

```
code/vars-on-stack.c

int add(int c, int d)

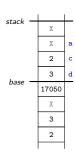
{

int a;

a = c + d;

return a;

}
```



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

```
code/vars-on-stack.c

int add(int c, int d)

{

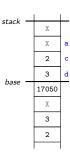
int a;

a = c + d;

return a;

}
```

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



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

```
code/vars-on-stack.c

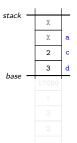
int add(int c, int d)

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

▶ Den anropade funktionen (add()) justerar stackpekarna...

```
code/vars-on-stack.c

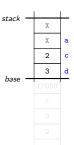
int add(int c, int d)

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

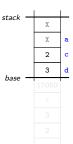
int add(int c, int d)

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

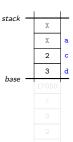
int add(int c, int d)

int a;

a = c + d;

return a;

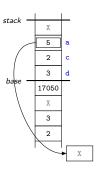
}
```



- ...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)

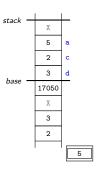
- ► Vid återhopp från en funktion sker följande:
 - returvärdet lagras i ett s.k. register i CPU:n,

```
code/vars-on-stack.c
     int add(int c, int d)
6
     {
          int a;
          a = c + d:
9
          return a;
10
11
     int main(void)
12
13
14
          int a = 2:
15
          int c = 3;
16
          int sum;
17
          sum = add(a, c);
```



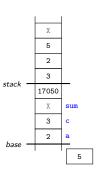
- ► Vid återhopp från en funktion sker följande:
 - returvärdet lagras i ett s.k. register i CPU:n,

```
code/vars-on-stack.c
     int add(int c, int d)
6
     {
          int a;
          a = c + d:
9
          return a;
10
11
     int main(void)
12
13
14
          int a = 2:
15
          int c = 3;
16
          int sum;
17
          sum = add(a, c);
```



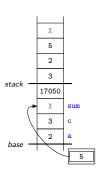
- ► Vid återhopp från en funktion sker följande:
 - returvärdet lagras i ett s.k. register i CPU:n,
 - stack frame återställs...

```
code/vars-on-stack.c -
     int add(int c, int d)
     {
          int a;
          a = c + d:
9
          return a;
10
11
     int main(void)
12
13
14
          int a = 2:
15
          int c = 3;
16
          int sum;
17
          sum = add(a, c);
```



- ► 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:
9
10
          return a;
11
     int main(void)
12
13
          int a = 2:
14
15
          int c = 3;
16
          int sum;
17
          sum = add(a, c);
```

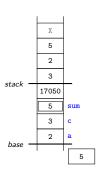


... exekveringen fortsätter vid återhoppsadressen...

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

- ► 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:
9
10
          return a;
11
     int main(void)
12
13
          int a = 2:
14
15
          int c = 3;
16
          int sum;
17
          sum = add(a, c);
```



... exekveringen fortsätter vid återhoppsadressen...

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

- 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:
9
10
          return a;
11
     int main(void)
12
13
          int a = 2:
14
15
          int c = 3;
16
          int sum;
17
          sum = add(a, c):
```

- ... exekveringen fortsätter vid återhoppsadressen...
 17050 store register at (base+8) // sum
- ...och sedan vidare på nästa rad...

```
code/vars-on-stack.c _
     #include <stdio.h>
1
     void print(int v)
3
     {
          printf("v = %d\n", v);
4
                                                                           X
5
                                                                           X
     int add(int c, int d)
6
     {
                                                                           X
          int a;
                                                                           X
          a = c + d;
9
                                                                           X
          return a:
10
                                                                           X
11
                                                                           X
     int main(void)
12
     {
                                                                           X
13
          int a = 2;
14
          int c = 3;
15
16
          int sum;
          sum = add(a, c);
17
          print(sum);
18
          sum = add(sum, c + 4);
19
          print(sum);
20
          return 0;
21
22
```

```
code/vars-on-stack.c
      #include <stdio.h>
 1
      void print(int v)
 3
      {
          printf("v = \frac{d}{n}",v);
 4
                                                                             X
 5
                                                                             X
      int add(int c, int d)
6
      {
                                                                             X
          int a;
                                                                             X
                                                                    stack
          a = c + d;
9
          return a:
10
                                                                                 sum
11
                                                                             X
      int main(void)
12
      }
                                                                             X
13
                                                                     base
          int a = 2;
14
          int c = 3;
15
16
          int sum;
          sum = add(a, c);
17
          print(sum);
18
          sum = add(sum, c + 4);
19
          print(sum);
20
          return 0;
21
22
```

```
code/vars-on-stack.c _
1
     #include <stdio.h>
     void print(int v)
3
     {
          printf("v = %d\n", v);
4
                                                                            X
5
                                                                            X
     int add(int c, int d)
6
     {
                                                                            X
          int a;
                                                                            X
                                                                   stack
          a = c + d;
9
          return a:
10
                                                                                sum
11
                                                                            X
     int main(void)
12
     {
                                                                            X
13
                                                                    base
          int a = 2;
14
          int c = 3:
15
16
          int sum;
          sum = add(a, c);
17
          print(sum);
18
          sum = add(sum, c + 4);
19
          print(sum);
20
          return 0;
21
22
```

```
code/vars-on-stack.c -
1
     #include <stdio.h>
     void print(int v)
3
     {
          printf("v = %d\n", v);
4
                                                                            X
5
                                                                            X
     int add(int c, int d)
6
     {
                                                                            X
          int a;
                                                                            X
                                                                   stack
          a = c + d;
9
          return a:
10
                                                                            X
                                                                                sum
11
                                                                            X
     int main(void)
12
     {
                                                                            2
13
                                                                    base
          int a = 2;
14
          int c = 3:
15
          int sum;
16
          sum = add(a, c);
17
          print(sum);
18
          sum = add(sum, c + 4);
19
          print(sum);
20
          return 0;
21
22
```

```
code/vars-on-stack.c _
1
     #include <stdio.h>
     void print(int v)
3
     {
          printf("v = %d\n", v);
4
                                                                            X
5
                                                                            X
     int add(int c, int d)
6
     {
                                                                            X
          int a;
                                                                            X
                                                                   stack
          a = c + d;
9
          return a:
10
                                                                                sum
11
                                                                            3
     int main(void)
12
     {
                                                                            2
13
                                                                    base
          int a = 2;
14
          int c = 3;
15
16
          int sum;
          sum = add(a, c);
17
          print(sum);
18
          sum = add(sum, c + 4);
19
          print(sum);
20
          return 0;
21
22
```

```
code/vars-on-stack.c
 1
      #include <stdio.h>
      void print(int v)
 3
      {
          printf("v = \frac{d}{n}",v);
 4
                                                                              X
 5
                                                                              X
      int add(int c, int d)
6
      {
                                                                              X
          int a;
                                                                              X
                                                                     stack
          a = c + d;
9
                                                                             17050
                                                                                  return address
          return a:
10
                                                                                  sum
11
                                                                              3
      int main(void)
12
      {
                                                                              2
13
                                                                      base .
          int a = 2;
14
          int c = 3;
15
16
          int sum;
          sum = add(a, c);
17
          print(sum);
18
          sum = add(sum, c + 4);
19
          print(sum);
20
          return 0;
21
22
```

```
code/vars-on-stack.c
 1
      #include <stdio.h>
      void print(int v)
 3
      {
          printf("v = %d\n",v);
 4
                                                                             X
 5
                                                                             X
      int add(int c, int d)
6
      {
                                                                             X
          int a;
                                                                                 parameter 2
                                                                             3
                                                                    stack
          a = c + d;
9
                                                                           17050
                                                                                 return address
          return a:
10
                                                                                 sum
11
                                                                             3
      int main(void)
12
      {
                                                                             2
13
                                                                     base
          int a = 2;
14
          int c = 3;
15
16
          int sum;
          sum = add(a, c);
17
          print(sum);
18
          sum = add(sum, c + 4);
19
          print(sum);
20
          return 0;
21
22
```

```
code/vars-on-stack.c
 1
      #include <stdio.h>
      void print(int v)
 3
      {
          printf("v = %d\n",v);
 4
                                                                              X
 5
                                                                              X
      int add(int c, int d)
6
      {
                                                                              2
                                                                                  parameter 1
          int a;
                                                                                  parameter 2
                                                                     stack
          a = c + d;
9
                                                                            17050
                                                                                  return address
          return a:
10
                                                                                  sum
11
                                                                              3
      int main(void)
12
      {
                                                                              2
13
                                                                     base
          int a = 2;
14
          int c = 3:
15
16
          int sum;
          sum = add(a, c);
17
          print(sum);
18
          sum = add(sum, c + 4);
19
          print(sum);
20
          return 0;
21
22
```

```
code/vars-on-stack.c
 1
      #include <stdio.h>
      void print(int v)
 3
      {
          printf("v = \frac{d}{n}",v);
                                                                    stack
 4
 5
                                                                             X
      int add(int c, int d)
6
          int a;
                                                                             3
                                                                     base
          a = c + d;
9
                                                                            17050
          return a:
10
                                                                             X
11
                                                                             3
      int main(void)
12
      {
                                                                             2
13
          int a = 2;
14
          int c = 3;
15
16
          int sum;
          sum = add(a, c);
17
          print(sum);
18
          sum = add(sum, c + 4);
19
          print(sum);
20
          return 0;
21
22
```

```
code/vars-on-stack.c
      #include <stdio.h>
 1
      void print(int v)
 3
      {
          printf("v = \frac{d}{n}",v);
                                                                     stack
 4
 5
                                                                             X
      int add(int c, int d)
6
      {
          int a;
                                                                             3
                                                                     base
          a = c + d;
9
                                                                            17050
          return a:
10
                                                                             X
11
                                                                             3
      int main(void)
12
      {
                                                                             2
13
          int a = 2;
14
          int c = 3;
15
16
          int sum;
          sum = add(a, c);
17
          print(sum);
18
          sum = add(sum, c + 4);
19
          print(sum);
20
          return 0;
21
22
```

```
code/vars-on-stack.c
      #include <stdio.h>
 1
      void print(int v)
 3
      {
          printf("v = \frac{d}{n}",v);
                                                                    stack
 4
 5
                                                                             5
      int add(int c, int d)
6
      {
          int a;
                                                                             3
                                                                     base
          a = c + d;
9
                                                                           17050
10
          return a:
                                                                             X
11
                                                                             3
      int main(void)
12
      {
                                                                             2
13
          int a = 2;
14
          int c = 3;
15
16
          int sum;
          sum = add(a, c);
17
          print(sum);
18
          sum = add(sum, c + 4);
19
          print(sum);
20
          return 0;
21
22
```

```
code/vars-on-stack.c -
 1
      #include <stdio.h>
      void print(int v)
 3
      {
          printf("v = %d\n", v);
                                                                    stack
 4
 5
                                                                            5
      int add(int c, int d)
6
      {
          int a;
                                                                            3
                                                                    base
          a = c + d;
9
                                                                           17050
          return a:
10
                                                                            X
11
                                                                            3
      int main(void)
12
      {
                                                                            2
13
          int a = 2;
14
                                                                                   5
          int c = 3;
15
16
          int sum;
          sum = add(a, c);
17
          print(sum);
18
          sum = add(sum, c + 4);
19
          print(sum);
20
          return 0;
21
22
```

```
code/vars-on-stack.c _
 1
      #include <stdio.h>
      void print(int v)
 3
      {
          printf("v = %d\n", v);
 4
                                                                            X
 5
                                                                            5
      int add(int c, int d)
6
      {
                                                                            2
          int a;
                                                                            3
                                                                    stack
          a = c + d;
9
                                                                           17050
          return a:
10
                                                                                sum
11
                                                                            3
      int main(void)
12
      {
                                                                            2
13
                                                                    base
          int a = 2;
14
                                                                                   5
          int c = 3;
15
16
          int sum;
          sum = add(a, c);
17
          print(sum);
18
          sum = add(sum, c + 4);
19
          print(sum);
20
          return 0;
21
22
```

```
code/vars-on-stack.c _
 1
      #include <stdio.h>
      void print(int v)
 3
      {
          printf("v = %d\n", v);
 4
                                                                             X
 5
                                                                             5
      int add(int c, int d)
6
      {
                                                                             2
          int a;
                                                                             3
                                                                    stack
          a = c + d;
9
                                                                           17050
          return a:
10
                                                                             5
                                                                                 sum
11
                                                                             3
      int main(void)
12
      {
                                                                             2
13
                                                                     base .
          int a = 2;
14
                                                                                   5
          int c = 3:
15
16
          int sum;
          sum = add(a, c);
17
          print(sum);
18
          sum = add(sum, c + 4);
19
          print(sum);
20
          return 0:
21
22
```

```
code/vars-on-stack.c
 1
      #include <stdio.h>
      void print(int v)
 3
      {
          printf("v = %d\n", v);
 4
                                                                             X
 5
                                                                              5
      int add(int c, int d)
6
      {
                                                                              2
          int a;
                                                                             3
                                                                     stack
          a = c + d;
9
                                                                            18050
                                                                                 return address
          return a:
10
                                                                                 sum
11
                                                                             3
      int main(void)
12
      {
                                                                             2
13
                                                                     base .
          int a = 2;
14
                                                                                    5
          int c = 3:
15
16
          int sum;
          sum = add(a, c);
17
          print(sum);
18
          sum = add(sum, c + 4);
19
          print(sum);
20
          return 0:
21
22
```

```
code/vars-on-stack.c
 1
      #include <stdio.h>
      void print(int v)
 3
      {
          printf("v = %d\n",v);
 4
                                                                              X
 5
                                                                              5
      int add(int c, int d)
6
      {
                                                                              2
          int a;
                                                                              5
                                                                                  parameter 1
                                                                     stack
          a = c + d;
9
                                                                            18050
                                                                                  return address
          return a:
10
                                                                                  sum
11
                                                                              3
      int main(void)
12
      {
                                                                              2
13
                                                                      base .
          int a = 2;
14
                                                                                    5
          int c = 3:
15
16
          int sum;
          sum = add(a, c);
17
          print(sum);
18
          sum = add(sum, c + 4);
19
          print(sum);
20
          return 0:
21
22
```

```
code/vars-on-stack.c -
 1
      #include <stdio.h>
      void print(int v)
      }
 3
          printf("v = %d\n", v);
 4
                                                                            X
 5
                                                                             5
      int add(int c, int d)
6
                                                                    stack
      {
          int a;
                                                                             5
                                                                    base
          a = c + d;
9
                                                                           18050
          return a:
10
                                                                            5
11
                                                                            3
      int main(void)
12
      {
                                                                            2
13
          int a = 2;
14
                                                                                   5
          int c = 3;
15
16
          int sum;
          sum = add(a, c);
17
          print(sum);
18
          sum = add(sum, c + 4);
19
          print(sum);
20
          return 0;
21
22
```

```
code/vars-on-stack.c _
1
     #include <stdio.h>
     void print(int v)
3
          printf("v = %d\n",v);
 4
                                                                            X
5
                                                                            5
     int add(int c, int d)
6
                                                                   stack
     {
          int a;
                                                                            5
                                                                    base
          a = c + d;
9
                                                                          18050
          return a:
10
                                                                            5
11
                                                                            3
     int main(void)
12
     {
                                                                            2
13
          int a = 2;
14
                                                                                  5
          int c = 3;
15
16
          int sum;
          sum = add(a, c);
17
          print(sum);
18
          sum = add(sum, c + 4);
19
          print(sum);
20
          return 0;
21
22
```

```
code/vars-on-stack.c
 1
      #include <stdio.h>
      void print(int v)
 3
      {
          printf("v = \frac{d}{n}",v);
 4
                                                                             X
 5
                                                                              5
      int add(int c, int d)
6
                                                                     stack
      {
          int a;
                                                                              5
                                                                     base
          a = c + d;
9
                                                                            18050
          return a:
10
                                                                             5
11
                                                                             3
      int main(void)
12
      {
                                                                             2
13
          int a = 2;
14
                                                                                    5
          int c = 3;
15
16
          int sum;
          sum = add(a, c);
17
          print(sum);
18
          sum = add(sum, c + 4);
19
          print(sum);
20
                                                     v = 5
          return 0;
21
22
```

```
code/vars-on-stack.c
      #include <stdio.h>
 1
      void print(int v)
 3
      {
          printf("v = \frac{d}{n}",v);
 4
                                                                              X
 5
                                                                              5
      int add(int c, int d)
6
      {
                                                                              2
          int a;
                                                                              5
                                                                     stack
          a = c + d;
9
                                                                            18050
          return a:
10
                                                                              5
                                                                                  sum
11
                                                                              3
      int main(void)
12
      {
                                                                              2
13
                                                                      base .
          int a = 2;
14
                                                                                    5
          int c = 3;
15
16
          int sum;
          sum = add(a, c);
17
          print(sum);
18
          sum = add(sum, c + 4);
19
          print(sum);
20
                                                     v = 5
          return 0:
21
22
```

```
code/vars-on-stack.c
 1
      #include <stdio.h>
      void print(int v)
 3
      {
          printf("v = %d\n",v);
 4
                                                                             X
 5
                                                                             5
      int add(int c, int d)
6
      {
                                                                             2
          int a;
                                                                             5
                                                                     stack
          a = c + d;
9
                                                                            19050
                                                                                 return address
          return a:
10
                                                                                 sum
11
                                                                             3
      int main(void)
12
      {
                                                                             2
13
                                                                     base .
          int a = 2;
14
                                                                                    5
          int c = 3;
15
16
          int sum;
          sum = add(a, c);
17
          print(sum);
18
          sum = add(sum, c + 4);
19
          print(sum);
20
                                                     v = 5
          return 0:
21
22
```

```
code/vars-on-stack.c
 1
      #include <stdio.h>
      void print(int v)
 3
      {
          printf("v = %d\n",v);
 4
                                                                              X
 5
                                                                              5
      int add(int c, int d)
6
      {
                                                                              2
          int a;
                                                                                  parameter 2
                                                                     stack
          a = c + d;
9
                                                                            19050
                                                                                  return address
          return a:
10
                                                                                  sum
11
                                                                              3
      int main(void)
12
      {
                                                                              2
13
                                                                      base .
          int a = 2;
14
                                                                                    5
          int c = 3:
15
16
          int sum;
          sum = add(a, c);
17
          print(sum);
18
          sum = add(sum, c + 4);
19
          print(sum);
20
                                                     v = 5
          return 0:
21
22
```

```
code/vars-on-stack.c
 1
      #include <stdio.h>
      void print(int v)
 3
      {
          printf("v = %d\n",v);
 4
                                                                              X
 5
                                                                               5
      int add(int c, int d)
6
      {
                                                                                  parameter 1
          int a;
                                                                                  parameter 2
                                                                     stack
          a = c + d;
9
                                                                             19050
                                                                                  return address
          return a:
10
                                                                                  sum
11
                                                                              3
      int main(void)
12
      {
                                                                              2
13
                                                                      base .
          int a = 2;
14
                                                                                     5
          int c = 3:
15
16
          int sum;
          sum = add(a, c);
17
          print(sum);
18
          sum = add(sum, c + 4);
19
          print(sum);
20
                                                      v = 5
          return 0:
21
22
```

```
code/vars-on-stack.c
 1
      #include <stdio.h>
      void print(int v)
 3
      {
          printf("v = \frac{d}{n}",v);
                                                                     stack
 4
 5
                                                                              5
      int add(int c, int d)
6
          int a;
                                                                     base
          a = c + d;
9
                                                                            19050
          return a:
10
                                                                             5
11
                                                                             3
      int main(void)
12
      {
                                                                             2
13
          int a = 2;
14
                                                                                    5
          int c = 3;
15
16
          int sum;
          sum = add(a, c);
17
          print(sum);
18
          sum = add(sum, c + 4);
19
          print(sum);
20
                                                     v = 5
          return 0;
21
22
```

```
code/vars-on-stack.c
      #include <stdio.h>
 1
      void print(int v)
 3
      {
          printf("v = %d\n", v);
                                                                    stack
 4
 5
                                                                            5
      int add(int c, int d)
6
      {
          int a;
                                                                            7
                                                                    base
          a = c + d;
9
                                                                           19050
          return a:
10
                                                                            5
11
                                                                            3
      int main(void)
12
      {
                                                                            2
13
          int a = 2;
14
                                                                                   5
          int c = 3;
15
16
          int sum;
          sum = add(a, c);
17
          print(sum);
18
          sum = add(sum, c + 4);
19
          print(sum);
20
                                                    v = 5
          return 0;
21
22
```

```
code/vars-on-stack.c
1
     #include <stdio.h>
     void print(int v)
3
     {
          printf("v = %d\n", v);
                                                                   stack
4
5
                                                                            12
     int add(int c, int d)
6
     {
          int a;
                                                                            7
                                                                    base
          a = c + d;
9
                                                                          19050
10
          return a:
                                                                            5
11
                                                                            3
     int main(void)
12
     {
                                                                            2
13
          int a = 2;
14
                                                                                  5
          int c = 3;
15
16
          int sum;
          sum = add(a, c);
17
          print(sum);
18
          sum = add(sum, c + 4);
19
          print(sum);
20
                                                    v = 5
          return 0;
21
22
```

```
code/vars-on-stack.c
     #include <stdio.h>
1
     void print(int v)
3
     {
          printf("v = %d\n", v);
                                                                   stack
4
5
                                                                            12
     int add(int c, int d)
6
     {
          int a;
                                                                            7
                                                                    base
          a = c + d;
9
                                                                          19050
          return a:
10
                                                                            5
11
                                                                            3
     int main(void)
12
     {
                                                                            2
13
          int a = 2;
14
                                                                                  12
          int c = 3;
15
16
          int sum;
          sum = add(a, c);
17
          print(sum);
18
          sum = add(sum, c + 4);
19
          print(sum);
20
                                                    v = 5
          return 0;
21
22
```

```
code/vars-on-stack.c
      #include <stdio.h>
 1
      void print(int v)
 3
      {
          printf("v = %d\n", v);
 4
                                                                             X
 5
                                                                            12
      int add(int c, int d)
6
      {
                                                                             5
          int a;
                                                                             7
                                                                    stack
          a = c + d;
9
                                                                           19050
          return a:
10
                                                                                 sum
11
                                                                             3
      int main(void)
12
      {
                                                                             2
13
                                                                    base .
          int a = 2;
14
                                                                                   12
          int c = 3;
15
16
          int sum;
          sum = add(a, c);
17
          print(sum);
18
          sum = add(sum, c + 4);
19
          print(sum);
20
                                                    v = 5
          return 0:
21
22
```

```
code/vars-on-stack.c
      #include <stdio.h>
 1
      void print(int v)
 3
      {
          printf("v = %d\n", v);
 4
                                                                             X
 5
                                                                            12
      int add(int c, int d)
6
      {
                                                                             5
          int a;
                                                                             7
                                                                    stack
          a = c + d;
9
                                                                           19050
          return a:
10
                                                                            12
                                                                                 sum
11
                                                                             3
      int main(void)
12
      {
                                                                             2
13
                                                                     base -
          int a = 2;
14
                                                                                   12
          int c = 3;
15
16
          int sum;
          sum = add(a, c);
17
          print(sum);
18
          sum = add(sum, c + 4);
19
          print(sum);
20
                                                    v = 5
21
          return 0:
22
```

```
code/vars-on-stack.c
 1
      #include <stdio.h>
      void print(int v)
 3
      {
          printf("v = %d\n", v);
 4
                                                                              X
 5
                                                                             12
      int add(int c, int d)
6
      {
                                                                              5
          int a;
                                                                              7
                                                                     stack
          a = c + d;
9
                                                                            20050
                                                                                  return address
          return a:
10
                                                                             12
                                                                                  sum
11
                                                                              3
      int main(void)
12
      {
                                                                              2
13
                                                                      base .
          int a = 2;
14
                                                                                    12
          int c = 3:
15
16
          int sum;
          sum = add(a, c);
17
          print(sum);
18
          sum = add(sum, c + 4);
19
          print(sum);
20
                                                     v = 5
21
          return 0:
22
```

```
code/vars-on-stack.c
 1
      #include <stdio.h>
      void print(int v)
 3
      {
          printf("v = %d\n",v);
 4
                                                                              X
 5
                                                                              12
      int add(int c, int d)
6
      {
                                                                              5
          int a;
                                                                                  parameter 1
                                                                              12
                                                                     stack
          a = c + d;
9
                                                                            20050
                                                                                  return address
          return a:
10
                                                                              12
                                                                                  sum
11
                                                                              3
      int main(void)
12
      {
                                                                              2
13
                                                                      base .
          int a = 2;
14
                                                                                    12
          int c = 3:
15
16
          int sum;
          sum = add(a, c);
17
          print(sum);
18
          sum = add(sum, c + 4);
19
          print(sum);
20
                                                     v = 5
21
          return 0:
22
```

```
code/vars-on-stack.c _
 1
      #include <stdio.h>
      void print(int v)
      }
 3
          printf("v = %d\n", v);
 4
                                                                            X
 5
                                                                            12
      int add(int c, int d)
6
                                                                    stack
      {
          int a;
                                                                            12
                                                                    base
          a = c + d;
9
                                                                           20050
          return a:
10
                                                                            12
11
                                                                            3
      int main(void)
12
      {
                                                                            2
13
          int a = 2;
14
                                                                                   12
          int c = 3;
15
16
          int sum;
          sum = add(a, c);
17
          print(sum);
18
          sum = add(sum, c + 4);
19
          print(sum);
20
                                                    v = 5
          return 0;
21
22
```

```
code/vars-on-stack.c
1
      #include <stdio.h>
     void print(int v)
3
          printf("v = %d\n",v);
 4
                                                                            X
5
                                                                            12
     int add(int c, int d)
6
                                                                   stack
     {
          int a;
                                                                            12
                                                                    base
          a = c + d;
9
                                                                          20050
          return a:
10
                                                                           12
11
                                                                            3
     int main(void)
12
     {
                                                                            2
13
          int a = 2;
14
                                                                                  12
          int c = 3;
15
16
          int sum;
          sum = add(a, c);
17
          print(sum);
18
          sum = add(sum, c + 4);
19
          print(sum);
20
                                                    v = 5
          return 0:
21
22
```

```
code/vars-on-stack.c
 1
      #include <stdio.h>
      void print(int v)
 3
      {
          printf("v = \frac{d}{n}",v);
 4
                                                                              X
 5
                                                                             12
      int add(int c, int d)
6
                                                                     stack
      {
          int a;
                                                                             12
                                                                     base
          a = c + d;
9
                                                                            20050
          return a:
10
                                                                             12
11
                                                                              3
      int main(void)
12
      {
                                                                              2
13
          int a = 2;
14
                                                                                    12
          int c = 3;
15
16
          int sum;
          sum = add(a, c);
17
          print(sum);
18
          sum = add(sum, c + 4);
19
          print(sum);
20
                                                     v = 5
          return 0;
21
                                                     v = 12
22
```

```
code/vars-on-stack.c
 1
      #include <stdio.h>
      void print(int v)
 3
      {
          printf("v = %d\n", v);
 4
                                                                             X
 5
                                                                             12
      int add(int c, int d)
6
      {
                                                                             5
          int a;
                                                                            12
                                                                    stack
          a = c + d;
9
                                                                           20050
          return a:
10
                                                                             12
                                                                                 sum
11
                                                                             3
      int main(void)
12
      {
                                                                             2
13
                                                                     base -
          int a = 2;
14
                                                                                   12
          int c = 3:
15
16
          int sum;
          sum = add(a, c);
17
          print(sum);
18
          sum = add(sum, c + 4);
19
          print(sum);
20
                                                     v = 5
          return 0:
21
                                                     v = 12
22
```

```
code/vars-on-stack.c
      #include <stdio.h>
 1
      void print(int v)
 3
      {
          printf("v = %d\n", v);
 4
                                                                             X
 5
                                                                             12
      int add(int c, int d)
6
      {
                                                                             5
          int a;
                                                                            12
                                                                    stack
          a = c + d;
9
                                                                           20050
          return a:
10
                                                                             12
                                                                                 sum
11
                                                                             3
      int main(void)
12
      {
                                                                             2
13
                                                                     base .
          int a = 2;
14
                                                                                   0
          int c = 3;
15
16
          int sum;
          sum = add(a, c);
17
          print(sum);
18
          sum = add(sum, c + 4);
19
          print(sum);
20
                                                     v = 5
          return 0;
21
                                                     v = 12
22
```

```
code/vars-on-stack.c
     #include <stdio.h>
1
     void print(int v)
3
     {
          printf("v = %d\n", v);
4
                                                                           X
5
                                                                           12
     int add(int c, int d)
6
     {
                                                                            5
          int a;
                                                                           12
          a = c + d;
9
                                                                          20050
          return a:
10
                                                                           12
11
                                                                           3
     int main(void)
12
     {
                                                                           2
13
          int a = 2;
14
          int c = 3:
15
16
          int sum;
          sum = add(a, c);
17
          print(sum);
18
          sum = add(sum, c + 4);
19
          print(sum);
20
                                                    v = 5
          return 0;
21
                                                    v = 12
22
```

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

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

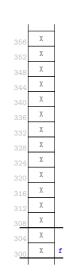
```
#include <stdio.h>
    int fib(int n)
 4
 5
         int fm1, fm2:
 6
         if (n < 2) {
             return n:
         } else {
            fm1 = fib(n - 1);
10
            fm2 = fib(n - 2);
11
            return fm1 + fm2:
12
         7
13
    1 }
14
15
    int main(void)
16
17
         int f = fib(3):
         printf("fib(3) = %d\n", f);
18
19
         return 0;
20
```

```
#include <stdio.h>
 23
     int fib(int n)
 4
     ł
 5
         int fm1, fm2;
         if (n < 2) {
             return n;
 8
         } else {
 9
             fm1 = fib(n - 1):
10
             fm2 = fib(n - 2);
11
            return fm1 + fm2;
12
13
14
15
     int main(void)
16
17
         int f = fib(3);
18
         printf("fib(3) = %d\n", f);
19
         return 0:
20
    }
```

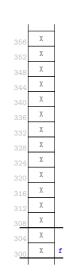
```
X
X
X
X
X
X
X
X
X
X
X
X
X
X
X
```



```
#include <stdio.h>
 23
     int fib(int n)
 4
     ł
 5
         int fm1, fm2;
         if (n < 2) {
             return n;
 8
         } else {
 9
             fm1 = fib(n - 1):
10
             fm2 = fib(n - 2);
11
             return fm1 + fm2;
12
13
14
15
     int main(void)
16
17
         int f = fib(3);
18
         printf("fib(3) = %d\n", f);
19
         return 0:
20
    }
```

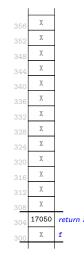


```
#include <stdio.h>
 23
     int fib(int n)
 4
     ł
 5
         int fm1, fm2;
         if (n < 2) {
             return n;
 8
         } else {
 9
             fm1 = fib(n - 1):
10
             fm2 = fib(n - 2);
11
             return fm1 + fm2;
12
13
     }
14
15
     int main(void)
16
17
         int f = fib(3);
18
         printf("fib(3) = %d\n", f);
19
         return 0:
20
    }
```





```
#include <stdio.h>
 23
     int fib(int n)
 4
     ł
 5
         int fm1, fm2;
         if (n < 2) {
             return n;
 8
         } else {
 9
             fm1 = fib(n - 1):
10
             fm2 = fib(n - 2);
11
             return fm1 + fm2;
12
13
     }
14
15
     int main(void)
16
17
         int f = fib(3);
18
         printf("fib(3) = %d\n", f);
19
         return 0:
20
    }
```



```
#include <stdio.h>
 2
 3
     int fib(int n)
 4
     ł
 5
         int fm1, fm2;
         if (n < 2) {
             return n;
 8
         } else {
 9
             fm1 = fib(n - 1):
10
             fm2 = fib(n - 2);
11
             return fm1 + fm2;
12
13
     }
14
15
     int main(void)
16
17
         int f = fib(3);
18
         printf("fib(3) = %d\n", f);
19
         return 0:
20
    }
```



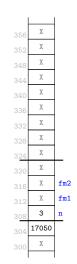
parame

return

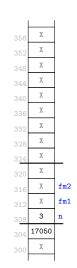
X



```
#include <stdio.h>
 23
     int fib(int n)
 4
 5
         int fm1, fm2;
 6
         if (n < 2) {
             return n;
 8
         } else {
 9
             fm1 = fib(n - 1):
10
             fm2 = fib(n - 2);
11
             return fm1 + fm2;
12
13
14
15
     int main(void)
16
     ſ
17
         int f = fib(3);
18
         printf("fib(3) = %d\n", f);
19
         return 0:
20
    }
```

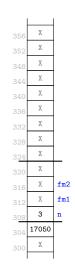


```
#include <stdio.h>
 2
 3
     int fib(int n)
 4
     ł
 5
         int fm1, fm2;
 6
         if (n < 2) {
 7
             return n;
 8
         } else {
 9
             fm1 = fib(n - 1):
10
             fm2 = fib(n - 2);
11
             return fm1 + fm2;
12
13
14
15
     int main(void)
16
     ſ
17
         int f = fib(3);
18
         printf("fib(3) = %d\n", f);
19
         return 0:
20
    }
```



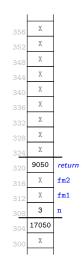


```
#include <stdio.h>
 2
 3
     int fib(int n)
 4
     ł
 5
         int fm1, fm2;
 6
         if (n < 2) {
 7
             return n;
 8
         } else {
9
             fm1 = fib(n - 1):
10
             fm2 = fib(n - 2);
11
             return fm1 + fm2;
12
13
14
15
     int main(void)
16
     ſ
17
         int f = fib(3);
18
         printf("fib(3) = %d\n", f);
19
         return 0:
20
    }
```





```
#include <stdio.h>
 2
 3
     int fib(int n)
 4
     ł
 5
         int fm1, fm2;
 6
         if (n < 2) {
             return n;
 8
         } else {
9
             fm1 = fib(n - 1):
10
             fm2 = fib(n - 2);
11
             return fm1 + fm2;
12
13
14
15
     int main(void)
16
17
         int f = fib(3);
18
         printf("fib(3) = %d\n", f);
19
         return 0:
20
    }
```



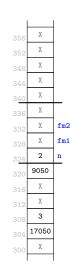
```
#include <stdio.h>
 2
 3
     int fib(int n)
 4
     ł
 5
         int fm1, fm2;
 6
         if (n < 2) {
 7
             return n;
 8
         } else {
9
             fm1 = fib(n - 1):
10
             fm2 = fib(n - 2);
11
             return fm1 + fm2;
12
13
14
15
     int main(void)
16
     ſ
17
         int f = fib(3);
18
         printf("fib(3) = %d\n", f);
19
         return 0:
20
    }
```



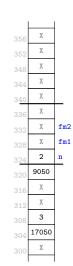
```
#include <stdio.h>
 23
     int fib(int n)
 4
 5
         int fm1, fm2;
 6
         if (n < 2) {
             return n;
 8
         } else {
 9
             fm1 = fib(n - 1):
10
             fm2 = fib(n - 2);
11
            return fm1 + fm2;
12
13
14
15
     int main(void)
16
17
         int f = fib(3);
18
         printf("fib(3) = %d\n", f);
19
         return 0:
20
    }
```



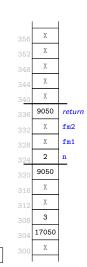
```
#include <stdio.h>
 2
 3
     int fib(int n)
 4
     ł
 5
         int fm1, fm2;
 6
         if (n < 2) {
 7
             return n;
 8
         } else {
 9
             fm1 = fib(n - 1):
10
             fm2 = fib(n - 2);
11
             return fm1 + fm2;
12
13
14
15
     int main(void)
16
17
         int f = fib(3);
18
         printf("fib(3) = %d\n", f);
19
         return 0:
20
    }
```



```
#include <stdio.h>
 2
 3
     int fib(int n)
 4
     ł
 5
         int fm1, fm2;
 6
         if (n < 2) {
             return n;
 8
         } else {
9
             fm1 = fib(n - 1):
10
             fm2 = fib(n - 2);
11
             return fm1 + fm2;
12
13
14
15
     int main(void)
16
17
         int f = fib(3);
18
         printf("fib(3) = %d\n", f);
19
         return 0:
20
    }
```



```
#include <stdio.h>
 2
 3
     int fib(int n)
 4
     ł
 5
         int fm1, fm2;
 6
         if (n < 2) {
             return n;
 8
         } else {
9
             fm1 = fib(n - 1):
10
             fm2 = fib(n - 2);
11
             return fm1 + fm2;
12
13
14
15
     int main(void)
16
17
         int f = fib(3);
18
         printf("fib(3) = %d\n", f);
19
         return 0:
20
    }
```



```
#include <stdio.h>
 2
 3
     int fib(int n)
 4
     ł
 5
         int fm1, fm2;
 6
         if (n < 2) {
             return n;
 8
         } else {
9
             fm1 = fib(n - 1):
10
             fm2 = fib(n - 2);
11
             return fm1 + fm2;
12
13
14
15
     int main(void)
16
17
         int f = fib(3);
18
         printf("fib(3) = %d\n", f);
19
         return 0:
20
    }
```



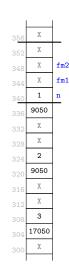
```
#include <stdio.h>
 23
     int fib(int n)
 4
 5
         int fm1, fm2;
 6
         if (n < 2) {
             return n;
 8
         } else {
 9
             fm1 = fib(n - 1):
10
             fm2 = fib(n - 2);
11
             return fm1 + fm2;
12
13
14
15
     int main(void)
16
17
         int f = fib(3);
18
         printf("fib(3) = %d\n", f);
19
         return 0:
20
    }
```



```
#include <stdio.h>
 2
 3
     int fib(int n)
 4
     ł
 5
         int fm1, fm2;
 6
         if (n < 2) {
 7
             return n;
 8
         } else {
 9
             fm1 = fib(n - 1):
10
             fm2 = fib(n - 2);
11
             return fm1 + fm2;
12
13
14
15
     int main(void)
16
17
         int f = fib(3);
18
         printf("fib(3) = %d\n", f);
19
         return 0:
20
    }
```



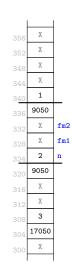
```
#include <stdio.h>
 2
 3
     int fib(int n)
 4
     ł
 5
         int fm1, fm2;
 6
         if (n < 2) {
 7
             return n;
 8
         } else {
 9
             fm1 = fib(n - 1):
10
             fm2 = fib(n - 2);
11
             return fm1 + fm2;
12
13
14
15
     int main(void)
16
17
         int f = fib(3);
18
         printf("fib(3) = %d\n", f);
19
         return 0:
20
    }
```



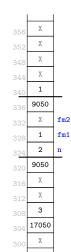
```
#include <stdio.h>
 2
 3
     int fib(int n)
 4
     ł
 5
         int fm1, fm2;
         if (n < 2) {
             return n;
 8
         } else {
 9
             fm1 = fib(n - 1):
10
             fm2 = fib(n - 2);
11
             return fm1 + fm2;
12
13
14
15
     int main(void)
16
17
         int f = fib(3);
18
         printf("fib(3) = %d\n", f);
19
         return 0:
20
    }
```



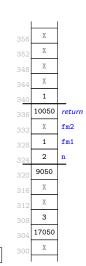
```
#include <stdio.h>
 2
 3
     int fib(int n)
 4
     ł
 5
         int fm1, fm2;
 6
         if (n < 2) {
             return n;
 8
         } else {
 9
             fm1 = fib(n - 1):
10
             fm2 = fib(n - 2);
11
             return fm1 + fm2;
12
13
14
15
     int main(void)
16
17
         int f = fib(3);
18
         printf("fib(3) = %d\n", f);
19
         return 0:
20
    }
```



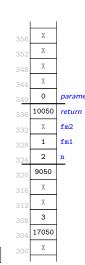
```
#include <stdio.h>
 2
 3
     int fib(int n)
 4
     ł
 5
         int fm1, fm2;
         if (n < 2) {
             return n;
 8
         } else {
 9
             fm1 = fib(n - 1):
10
             fm2 = fib(n - 2);
11
             return fm1 + fm2;
12
13
14
15
     int main(void)
16
17
         int f = fib(3);
18
         printf("fib(3) = %d\n", f);
19
         return 0:
20
    }
```



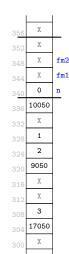
```
#include <stdio.h>
 2
 3
     int fib(int n)
 4
     ł
 5
         int fm1, fm2;
         if (n < 2) {
             return n;
 8
         } else {
 9
             fm1 = fib(n - 1):
10
             fm2 = fib(n - 2);
11
             return fm1 + fm2;
12
13
14
15
     int main(void)
16
17
         int f = fib(3);
18
         printf("fib(3) = %d\n", f);
19
         return 0:
20
    }
```



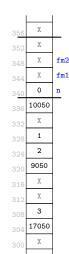
```
#include <stdio.h>
 2
 3
     int fib(int n)
 4
     ł
 5
         int fm1, fm2;
         if (n < 2) {
             return n;
 8
         } else {
 9
             fm1 = fib(n - 1):
10
             fm2 = fib(n - 2);
11
             return fm1 + fm2;
12
13
14
15
     int main(void)
16
17
         int f = fib(3);
18
         printf("fib(3) = %d\n", f);
19
         return 0:
20
    }
```



```
#include <stdio.h>
 23
     int fib(int n)
 4
 5
         int fm1, fm2;
 6
         if (n < 2) {
             return n;
 8
         } else {
 9
             fm1 = fib(n - 1):
10
             fm2 = fib(n - 2);
11
             return fm1 + fm2;
12
13
14
15
     int main(void)
16
17
         int f = fib(3);
18
         printf("fib(3) = %d\n", f);
19
         return 0:
20
    }
```



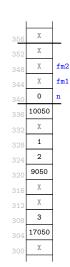
```
#include <stdio.h>
 2
 3
     int fib(int n)
 4
     ł
 5
         int fm1, fm2;
 6
         if (n < 2) {
 7
             return n;
 8
         } else {
 9
             fm1 = fib(n - 1):
10
             fm2 = fib(n - 2);
11
             return fm1 + fm2;
12
13
14
15
     int main(void)
16
17
         int f = fib(3);
18
         printf("fib(3) = %d\n", f);
19
         return 0:
20
    }
```



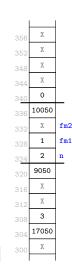
```
#include <stdio.h>
 2
 3
     int fib(int n)
 4
     ł
 5
         int fm1, fm2;
 6
         if (n < 2) {
 7
             return n;
 8
         } else {
 9
             fm1 = fib(n - 1):
10
             fm2 = fib(n - 2);
11
             return fm1 + fm2;
12
13
14
15
     int main(void)
16
17
         int f = fib(3);
18
         printf("fib(3) = %d\n", f);
19
         return 0:
20
    }
```



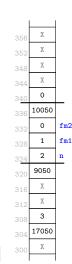
```
#include <stdio.h>
 2
 3
     int fib(int n)
 4
     ł
 5
         int fm1, fm2;
         if (n < 2) {
             return n;
 8
         } else {
 9
             fm1 = fib(n - 1):
10
             fm2 = fib(n - 2);
11
             return fm1 + fm2;
12
13
14
15
     int main(void)
16
17
         int f = fib(3);
18
         printf("fib(3) = %d\n", f);
19
         return 0:
20
    }
```



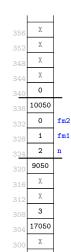
```
#include <stdio.h>
 2
 3
     int fib(int n)
 4
     ł
 5
         int fm1, fm2;
         if (n < 2) {
             return n;
 8
         } else {
 9
             fm1 = fib(n - 1):
10
             fm2 = fib(n - 2);
11
             return fm1 + fm2;
12
13
14
15
     int main(void)
16
17
         int f = fib(3);
18
         printf("fib(3) = %d\n", f);
19
         return 0:
20
    }
```



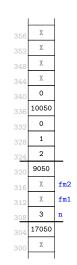
```
#include <stdio.h>
 2
 3
     int fib(int n)
 4
     ł
 5
         int fm1, fm2;
         if (n < 2) {
             return n;
 8
         } else {
 9
             fm1 = fib(n - 1):
10
             fm2 = fib(n - 2);
11
             return fm1 + fm2;
12
13
14
15
     int main(void)
16
17
         int f = fib(3);
18
         printf("fib(3) = %d\n", f);
19
         return 0:
20
    }
```



```
#include <stdio.h>
 2
 3
     int fib(int n)
 4
     ł
 5
         int fm1, fm2;
         if (n < 2) {
             return n;
 8
         } else {
 9
             fm1 = fib(n - 1):
10
             fm2 = fib(n - 2);
11
            return fm1 + fm2;
12
13
14
15
     int main(void)
16
17
         int f = fib(3);
18
         printf("fib(3) = %d\n", f);
19
         return 0:
20
    }
```

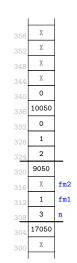


```
#include <stdio.h>
 2
 3
     int fib(int n)
 4
     ł
 5
         int fm1, fm2;
 6
         if (n < 2) {
             return n;
 8
         } else {
 9
             fm1 = fib(n - 1):
10
             fm2 = fib(n - 2);
11
             return fm1 + fm2;
12
13
14
15
     int main(void)
16
17
         int f = fib(3);
18
         printf("fib(3) = %d\n", f);
19
         return 0:
20
    }
```

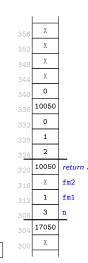




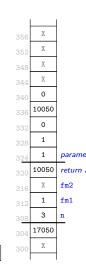
```
#include <stdio.h>
 2
 3
     int fib(int n)
 4
     ł
 5
         int fm1, fm2;
         if (n < 2) {
             return n;
 8
         } else {
 9
             fm1 = fib(n - 1):
10
             fm2 = fib(n - 2);
11
             return fm1 + fm2;
12
13
14
15
     int main(void)
16
17
         int f = fib(3);
18
         printf("fib(3) = %d\n", f);
19
         return 0:
20
    }
```



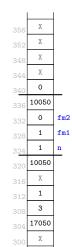
```
#include <stdio.h>
 2
 3
     int fib(int n)
 4
     ł
 5
         int fm1, fm2;
         if (n < 2) {
             return n;
 8
         } else {
 9
             fm1 = fib(n - 1):
10
             fm2 = fib(n - 2);
11
             return fm1 + fm2;
12
13
14
15
     int main(void)
16
17
         int f = fib(3);
18
         printf("fib(3) = %d\n", f);
19
         return 0:
20
    }
```



```
#include <stdio.h>
 2
 3
     int fib(int n)
 4
     ł
 5
         int fm1, fm2;
         if (n < 2) {
             return n;
 8
         } else {
 9
             fm1 = fib(n - 1):
10
             fm2 = fib(n - 2);
11
             return fm1 + fm2;
12
13
14
15
     int main(void)
16
17
         int f = fib(3);
18
         printf("fib(3) = %d\n", f);
19
         return 0:
20
    }
```



```
#include <stdio.h>
 23
     int fib(int n)
 4
 5
         int fm1, fm2;
 6
         if (n < 2) {
             return n;
 8
         } else {
 9
             fm1 = fib(n - 1):
10
             fm2 = fib(n - 2);
11
             return fm1 + fm2;
12
13
14
15
     int main(void)
16
17
         int f = fib(3);
18
         printf("fib(3) = %d\n", f);
19
         return 0:
20
    }
```

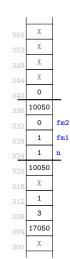




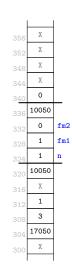




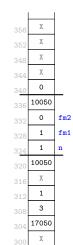
```
#include <stdio.h>
 2
 3
     int fib(int n)
 4
     ł
 5
         int fm1, fm2;
 6
         if (n < 2) {
 7
             return n;
 8
         } else {
 9
             fm1 = fib(n - 1):
10
             fm2 = fib(n - 2);
11
             return fm1 + fm2;
12
13
14
15
     int main(void)
16
17
         int f = fib(3);
18
         printf("fib(3) = %d\n", f);
19
         return 0:
20
    }
```



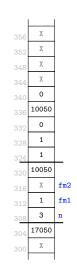
```
#include <stdio.h>
 2
 3
     int fib(int n)
 4
     ł
 5
         int fm1, fm2;
 6
         if (n < 2) {
 7
             return n;
 8
         } else {
 9
             fm1 = fib(n - 1):
10
             fm2 = fib(n - 2);
11
             return fm1 + fm2;
12
13
14
15
     int main(void)
16
17
         int f = fib(3);
18
         printf("fib(3) = %d\n", f);
19
         return 0:
20
    }
```



```
#include <stdio.h>
 2
 3
     int fib(int n)
 4
     ł
 5
         int fm1, fm2;
         if (n < 2) {
             return n;
 8
         } else {
 9
             fm1 = fib(n - 1):
10
             fm2 = fib(n - 2);
11
            return fm1 + fm2;
12
13
14
15
     int main(void)
16
17
         int f = fib(3);
18
         printf("fib(3) = %d\n", f);
19
         return 0:
20
    }
```

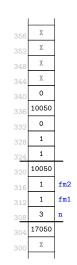


```
#include <stdio.h>
 2
 3
     int fib(int n)
 4
     ł
 5
         int fm1, fm2;
         if (n < 2) {
             return n;
 8
         } else {
 9
             fm1 = fib(n - 1):
10
             fm2 = fib(n - 2);
11
             return fm1 + fm2;
12
13
14
15
     int main(void)
16
17
         int f = fib(3);
18
         printf("fib(3) = %d\n", f);
19
         return 0:
20
    }
```



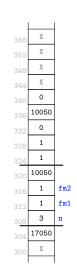


```
#include <stdio.h>
 2
 3
     int fib(int n)
 4
     ł
 5
         int fm1, fm2;
         if (n < 2) {
             return n;
 8
         } else {
 9
             fm1 = fib(n - 1):
10
             fm2 = fib(n - 2);
11
             return fm1 + fm2;
12
13
14
15
     int main(void)
16
17
         int f = fib(3);
18
         printf("fib(3) = %d\n", f);
19
         return 0:
20
    }
```

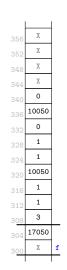




```
#include <stdio.h>
 2
 3
     int fib(int n)
 4
     ł
 5
         int fm1, fm2;
         if (n < 2) {
             return n;
 8
         } else {
 9
             fm1 = fib(n - 1):
10
             fm2 = fib(n - 2);
11
             return fm1 + fm2;
12
13
14
15
     int main(void)
16
17
         int f = fib(3);
18
         printf("fib(3) = %d\n", f);
19
         return 0:
20
    }
```

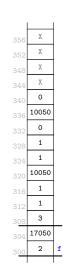


```
#include <stdio.h>
 2
 3
     int fib(int n)
 4
     ł
 5
         int fm1, fm2;
         if (n < 2) {
             return n;
 8
         } else {
 9
             fm1 = fib(n - 1):
10
             fm2 = fib(n - 2);
11
             return fm1 + fm2;
12
13
14
15
     int main(void)
16
17
         int f = fib(3);
18
         printf("fib(3) = %d\n", f);
19
         return 0:
20
    }
```

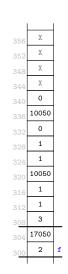




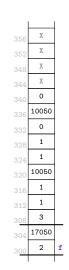
```
#include <stdio.h>
 2
 3
     int fib(int n)
 4
     ł
 5
         int fm1, fm2;
         if (n < 2) {
             return n;
 8
         } else {
 9
             fm1 = fib(n - 1):
10
             fm2 = fib(n - 2);
11
             return fm1 + fm2;
12
13
14
15
     int main(void)
16
17
         int f = fib(3);
18
         printf("fib(3) = %d\n", f);
19
         return 0:
20
    }
```



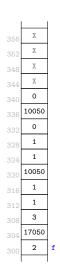
```
#include <stdio.h>
 2
 3
     int fib(int n)
 4
     ł
 5
         int fm1, fm2;
         if (n < 2) {
             return n;
 8
         } else {
 9
             fm1 = fib(n - 1):
10
             fm2 = fib(n - 2);
11
             return fm1 + fm2;
12
13
14
15
     int main(void)
16
17
         int f = fib(3);
18
         printf("fib(3) = %d\n", f);
19
         return 0;
20
```



```
#include <stdio.h>
 2
 3
     int fib(int n)
 4
     ł
 5
         int fm1, fm2;
         if (n < 2) {
             return n;
 8
         } else {
 9
             fm1 = fib(n - 1):
10
             fm2 = fib(n - 2);
11
            return fm1 + fm2;
12
13
14
15
     int main(void)
16
17
         int f = fib(3);
18
         printf("fib(3) = %d\n", f);
19
         return 0;
20
```



```
#include <stdio.h>
 2
 3
     int fib(int n)
 4
     ł
 5
         int fm1, fm2;
         if (n < 2) {
             return n;
 8
         } else {
 9
             fm1 = fib(n - 1):
10
             fm2 = fib(n - 2);
11
             return fm1 + fm2;
12
13
14
15
     int main(void)
16
17
         int f = fib(3);
18
         printf("fib(3) = %d\n", f);
19
         return 0:
20
    }
```



```
_ code/fib.c -
     #include <stdio.h>
1
     int fib(int n)
3
         int fm1, fm2;
5
         if (n < 2) {
6
              return n;
         } else {
              fm1 = fib(n - 1);
9
              fm2 = fib(n - 2);
10
              return fm1 + fm2;
11
12
     }
13
14
     int main(void)
15
16
17
          int f = fib(3);
         printf("fib(3) = %d\n", f);
18
19
         return 0;
20
```

main

```
__ code/fib.c -
     #include <stdio.h>
1
     int fib(int n)
3
         int fm1, fm2;
5
         if (n < 2) {
6
              return n;
         } else {
              fm1 = fib(n - 1);
9
              fm2 = fib(n - 2);
10
              return fm1 + fm2;
11
12
     }
13
14
     int main(void)
15
16
         int f = fib(3);
17
         printf("fib(3) = %d\n", f);
18
19
         return 0;
20
```

main

```
__ code/fib.c -
     #include <stdio.h>
1
     int fib(int n)
3
         int fm1, fm2;
5
         if (n < 2) {
6
              return n;
         } else {
              fm1 = fib(n - 1);
9
              fm2 = fib(n - 2);
10
              return fm1 + fm2;
11
12
     }
13
14
     int main(void)
15
16
          int f = fib(3);
17
         printf("fib(3) = %d\n", f);
18
19
         return 0;
20
```

```
main
```

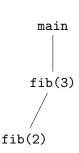
```
__ code/fib.c -
     #include <stdio.h>
1
     int fib(int n)
3
         int fm1, fm2;
5
         if (n < 2) {
6
              return n;
         } else {
              fm1 = fib(n - 1);
9
              fm2 = fib(n - 2);
10
              return fm1 + fm2;
11
12
     }
13
14
     int main(void)
15
     {
16
17
         int f = fib(3);
         printf("fib(3) = %d\n", f);
18
19
         return 0;
20
```

```
main | fib(3)
```

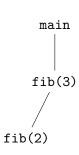
```
__ code/fib.c -
     #include <stdio.h>
1
     int fib(int n)
3
         int fm1, fm2;
5
          if (n < 2) {
6
              return n;
          } else {
              fm1 = fib(n - 1);
9
              fm2 = fib(n - 2);
10
              return fm1 + fm2;
11
12
     }
13
14
     int main(void)
15
     {
16
17
         int f = fib(3);
         printf("fib(3) = %d\n", f);
18
19
         return 0;
20
```

```
main | fib(3)
```

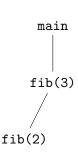
```
_ code/fib.c -
     #include <stdio.h>
1
     int fib(int n)
3
         int fm1, fm2;
5
          if (n < 2) {
6
              return n;
          } else {
              fm1 = fib(n - 1);
9
              fm2 = fib(n - 2);
10
              return fm1 + fm2;
11
12
     }
13
14
     int main(void)
15
     {
16
17
         int f = fib(3);
         printf("fib(3) = %d\n", f);
18
19
         return 0;
20
```



```
__ code/fib.c -
     #include <stdio.h>
1
     int fib(int n)
3
         int fm1, fm2;
5
         if (n < 2) {
6
              return n;
         } else {
              fm1 = fib(n - 1);
9
              fm2 = fib(n - 2);
10
              return fm1 + fm2;
11
12
     }
13
14
     int main(void)
15
     {
16
17
         int f = fib(3);
         printf("fib(3) = %d\n", f);
18
19
         return 0;
20
```



```
_ code/fib.c -
     #include <stdio.h>
1
     int fib(int n)
3
         int fm1, fm2;
5
          if (n < 2) {
6
              return n;
          } else {
              fm1 = fib(n - 1);
9
              fm2 = fib(n - 2);
10
              return fm1 + fm2;
11
12
     }
13
14
     int main(void)
15
     {
16
17
         int f = fib(3);
         printf("fib(3) = %d\n", f);
18
19
         return 0;
20
```



1

3

5

6

9

10

11 12

13 14

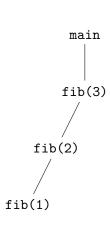
15

16 17

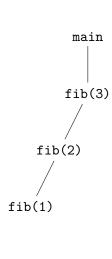
18 19

20

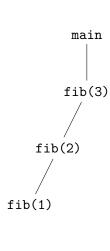
```
_ code/fib.c -
#include <stdio.h>
int fib(int n)
    int fm1, fm2;
    if (n < 2) {
        return n;
    } else {
        fm1 = fib(n - 1);
        fm2 = fib(n - 2);
        return fm1 + fm2;
}
int main(void)
{
    int f = fib(3);
    printf("fib(3) = %d\n", f);
    return 0;
```



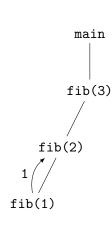
```
_ code/fib.c -
     #include <stdio.h>
1
     int fib(int n)
3
         int fm1, fm2;
5
         if (n < 2) {
6
              return n;
         } else {
              fm1 = fib(n - 1);
9
              fm2 = fib(n - 2);
10
              return fm1 + fm2;
11
12
     }
13
14
     int main(void)
15
     {
16
17
         int f = fib(3);
         printf("fib(3) = %d\n", f);
18
19
         return 0;
20
```



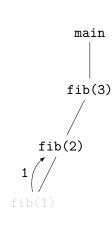
```
_ code/fib.c -
     #include <stdio.h>
1
     int fib(int n)
3
         int fm1, fm2;
5
          if (n < 2) {
6
              return n;
         } else {
              fm1 = fib(n - 1);
9
              fm2 = fib(n - 2);
10
              return fm1 + fm2;
11
12
     }
13
14
     int main(void)
15
     {
16
17
         int f = fib(3);
         printf("fib(3) = %d\n", f);
18
19
         return 0;
20
```



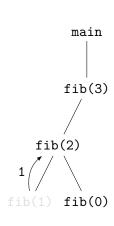
```
_ code/fib.c .
     #include <stdio.h>
1
     int fib(int n)
3
         int fm1, fm2;
5
          if (n < 2) {
6
              return n;
         } else {
              fm1 = fib(n - 1);
9
              fm2 = fib(n - 2);
10
              return fm1 + fm2;
11
12
     }
13
14
     int main(void)
15
     {
16
17
         int f = fib(3);
         printf("fib(3) = %d\n", f);
18
19
         return 0;
20
```



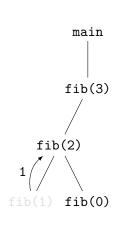
```
_ code/fib.c .
     #include <stdio.h>
1
     int fib(int n)
3
         int fm1, fm2;
5
         if (n < 2) {
6
              return n;
          } else {
              fm1 = fib(n - 1);
9
              fm2 = fib(n - 2);
10
              return fm1 + fm2;
11
12
     }
13
14
     int main(void)
15
     {
16
17
         int f = fib(3);
         printf("fib(3) = %d\n", f);
18
19
         return 0;
20
```



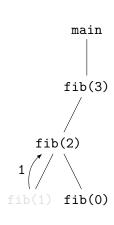
```
code/fib.c
     #include <stdio.h>
1
     int fib(int n)
3
         int fm1, fm2;
5
          if (n < 2) {
6
              return n;
          } else {
              fm1 = fib(n - 1);
9
              fm2 = fib(n - 2);
10
              return fm1 + fm2;
11
12
     }
13
14
     int main(void)
15
     {
16
17
         int f = fib(3);
         printf("fib(3) = %d\n", f);
18
19
         return 0;
20
```



```
code/fib.c
     #include <stdio.h>
1
     int fib(int n)
3
         int fm1, fm2;
5
          if (n < 2) {
6
              return n;
         } else {
              fm1 = fib(n - 1);
9
              fm2 = fib(n - 2);
10
              return fm1 + fm2;
11
12
     }
13
14
     int main(void)
15
     {
16
17
         int f = fib(3);
         printf("fib(3) = %d\n", f);
18
19
         return 0;
20
```



```
code/fib.c
     #include <stdio.h>
1
     int fib(int n)
3
         int fm1, fm2;
5
          if (n < 2) {
6
              return n;
          } else {
              fm1 = fib(n - 1);
9
              fm2 = fib(n - 2);
10
              return fm1 + fm2;
11
12
     }
13
14
     int main(void)
15
     {
16
17
         int f = fib(3);
         printf("fib(3) = %d\n", f);
18
19
         return 0;
20
```



1

3

5

6

9

10

11 12

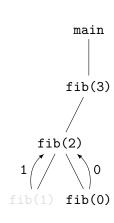
13 14

15

16 17

18 19

```
code/fib.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

3

5

6

9

10

11

12

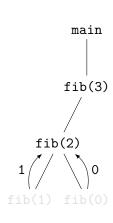
13 14

15

16 17

18 19

```
code/fib.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

3

5

6

9

10

11

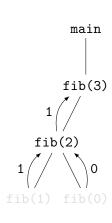
12

13 14 15

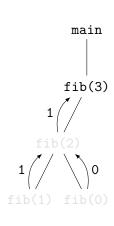
16 17

18 19

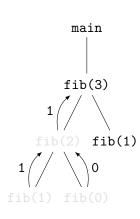
```
code/fib.c
#include <stdio.h>
int fib(int n)
    int fm1, fm2;
    if (n < 2) {
        return n;
    } else {
        fm1 = fib(n - 1);
        fm2 = fib(n - 2);
        return fm1 + fm2;
    }
}
int main(void)
{
    int f = fib(3);
    printf("fib(3) = %d\n", f);
    return 0;
```



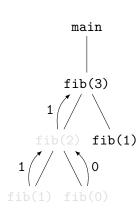
```
_ code/fib.c .
     #include <stdio.h>
1
     int fib(int n)
3
         int fm1, fm2;
5
         if (n < 2) {
6
              return n;
          } else {
              fm1 = fib(n - 1);
9
              fm2 = fib(n - 2);
10
              return fm1 + fm2;
11
12
     }
13
14
15
     int main(void)
     {
16
17
         int f = fib(3);
         printf("fib(3) = %d\n", f);
18
19
         return 0;
20
```



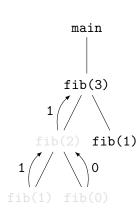
```
code/fib.c
     #include <stdio.h>
1
     int fib(int n)
3
         int fm1, fm2;
5
          if (n < 2) {
6
              return n;
          } else {
              fm1 = fib(n - 1);
9
              fm2 = fib(n - 2);
10
              return fm1 + fm2;
11
12
     }
13
14
     int main(void)
15
     {
16
17
         int f = fib(3);
         printf("fib(3) = %d\n", f);
18
19
         return 0;
20
```



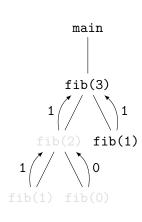
```
code/fib.c
     #include <stdio.h>
1
     int fib(int n)
3
         int fm1, fm2;
5
          if (n < 2) {
6
              return n;
         } else {
              fm1 = fib(n - 1);
9
              fm2 = fib(n - 2);
10
              return fm1 + fm2;
11
12
     }
13
14
15
     int main(void)
     {
16
17
         int f = fib(3);
         printf("fib(3) = %d\n", f);
18
19
         return 0;
20
```



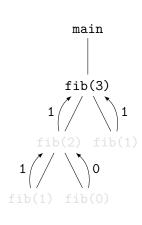
```
code/fib.c
     #include <stdio.h>
1
     int fib(int n)
3
         int fm1, fm2;
5
          if (n < 2) {
6
              return n;
          } else {
              fm1 = fib(n - 1);
9
              fm2 = fib(n - 2);
10
              return fm1 + fm2;
11
12
     }
13
14
15
     int main(void)
     {
16
17
         int f = fib(3);
         printf("fib(3) = %d\n", f);
18
19
         return 0;
20
```



```
code/fib.c
     #include <stdio.h>
1
     int fib(int n)
3
         int fm1, fm2;
5
          if (n < 2) {
6
              return n;
          } else {
              fm1 = fib(n - 1);
9
              fm2 = fib(n - 2);
10
              return fm1 + fm2;
11
12
     }
13
14
15
     int main(void)
     {
16
17
         int f = fib(3);
         printf("fib(3) = %d\n", f);
18
19
         return 0;
20
```



```
code/fib.c
     #include <stdio.h>
1
     int fib(int n)
3
         int fm1, fm2;
5
          if (n < 2) {
6
              return n;
         } else {
              fm1 = fib(n - 1);
9
              fm2 = fib(n - 2);
10
              return fm1 + fm2;
11
         }
12
     }
13
14
15
     int main(void)
     {
16
17
         int f = fib(3);
         printf("fib(3) = %d\n", f);
18
19
         return 0;
20
```



1

3

5

6

9

10

11

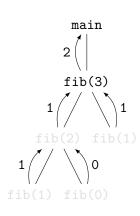
12

13 14 15

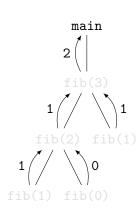
16 17

18 19

```
code/fib.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;
```



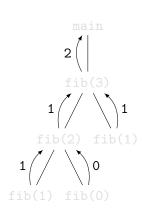
```
code/fib.c.
     #include <stdio.h>
1
     int fib(int n)
3
         int fm1, fm2;
5
          if (n < 2) {
6
              return n;
         } else {
              fm1 = fib(n - 1);
9
              fm2 = fib(n - 2);
10
              return fm1 + fm2;
11
12
     }
13
14
15
     int main(void)
16
17
          int f = fib(3);
          printf("fib(3) = %d\n", f);
18
19
         return 0;
20
```



```
code/fib.c -
     #include <stdio.h>
1
     int fib(int n)
3
         int fm1, fm2;
5
          if (n < 2) {
6
              return n;
         } else {
              fm1 = fib(n - 1);
9
              fm2 = fib(n - 2);
10
              return fm1 + fm2;
11
12
     }
13
14
15
     int main(void)
16
17
          int f = fib(3);
          printf("fib(3) = %d\n", f);
18
19
         return 0;
20
```

```
main
```

```
code/fib.c.
     #include <stdio.h>
1
     int fib(int n)
3
         int fm1, fm2;
5
          if (n < 2) {
6
              return n;
         } else {
              fm1 = fib(n - 1);
9
              fm2 = fib(n - 2);
10
              return fm1 + fm2;
11
12
     }
13
14
15
     int main(void)
     {
16
17
         int f = fib(3);
         printf("fib(3) = %d\n", f);
18
19
         return 0;
20
```



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
     int main(void)
12
     {
13
         int a = 2;
         int c = 3;
         int sum;
```

- 14 15 16
- ▶ 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)

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

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



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

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



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

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



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

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



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

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

312 X 308 304 Q 304 300 P 300 X i

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



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

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



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



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

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



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



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

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



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



► 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;
```



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



► 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;
```



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

► 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;
```



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



► 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

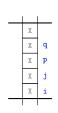
```
int i, *p, **q;
p = &i;
q = &p;
```



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

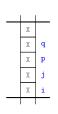


```
int i, j, *p, *q;
p = &i;
q = &i;
*p = 4; // Same effect as i=4
j = *q; // Same effect as j=i
```



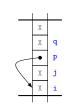
- Om p pekar på (refererar till) variabeln i så kan vi dereferera p för att komma åt värdet i i
 - ► Det kallas ibland att vi följer pekaren
- Det görs genom att skriva en stjärna framför variabelnamnet
 - ► Om p är av typen int * så är uttrycket *p av typen int
- Att p och q pekar på samma variabel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

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



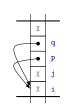
- Om p pekar på (refererar till) variabeln i så kan vi dereferera p för att komma åt värdet i i
 - ► Det kallas ibland att vi följer pekaren
- Det görs genom att skriva en stjärna framför variabelnamnet
 - Om p är av typen int * så är uttrycket *p av typen int
- Att p och q pekar på samma variabel kallas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

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



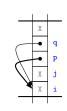
- Om p pekar på (refererar till) variabeln i så kan vi dereferera p för att komma åt värdet i i
 - Det kallas ibland att vi följer pekaren
- Det görs genom att skriva en stjärna framför variabelnamnet
 - Om p är av typen int * så är uttrycket *p av typen int
- Att p och q pekar på samma variabel kallas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

```
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

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

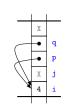


- Om p pekar på (refererar till) variabeln i så kan vi dereferera p för att komma åt värdet i i
 - ► Det kallas ibland att vi följer pekaren
- Det görs genom att skriva en stjärna framför variabelnamnet
 - ► Om p är av typen int * så är uttrycket *p av typen int
- Att p och q pekar på samma variabel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

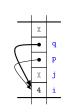


- Om p pekar på (refererar till) variabeln i så kan vi dereferera p för att komma åt värdet i i
 - ► Det kallas ibland att vi följer pekaren
- Det görs genom att skriva en stjärna framför variabelnamnet
 - ► Om p är av typen int * så är uttrycket *p av typen int
- Att p och q pekar på samma variabel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

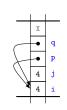


- Om p pekar på (refererar till) variabeln i så kan vi dereferera p för att komma åt värdet i i
 - ► Det kallas ibland att vi följer pekaren
- Det görs genom att skriva en stjärna framför variabelnamnet
 - ► Om p är av typen int * så är uttrycket *p av typen int
- Att p och q pekar på samma variabel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

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

Vad kommer att skrivas ut av den här koden?

```
_ code/add-one1.c ___
     #include <stdio.h>
1
     void add_one(int n)
2
3
         n = n + 1;
4
5
     int main(void)
6
7
          int a = 5:
8
          add_one(a);
          printf("a = %d\n", a);
10
11
          return 0;
12
```

...och den här?

```
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;
}
```

```
code/add-one1.c ____
     #include <stdio.h>
     void add_one(int n)
3
         n = n + 1;
4
     int main(void)
     {
         int a = 5;
8
         add_one(a);
9
         printf("a = %d\n", a);
10
         return 0;
11
12
```

```
X X X X X
```

```
code/add-one1.c ___
     #include <stdio.h>
                                                                     X
     void add_one(int n)
                                                                     X
                                                             stack ·
3
         n = n + 1;
4
                                                                     X
                                                             base
     int main(void)
     }
7
          int a = 5;
8
          add_one(a);
9
          printf("a = %d\n", a);
10
          return 0;
11
12
```

```
code/add-one1.c ____
     #include <stdio.h>
                                                                     X
     void add_one(int n)
                                                                     X
                                                             stack ·
3
         n = n + 1;
4
                                                                     X
                                                             base
     int main(void)
7
          int a = 5;
8
9
          add_one(a);
          printf("a = %d\n", a);
10
          return 0;
11
12
```

```
code/add-one1.c ____
     #include <stdio.h>
                                                                     X
     void add_one(int n)
                                                                     X
                                                             stack ·
3
         n = n + 1;
4
                                                                     5
                                                             base
     int main(void)
     {
7
8
          int a = 5;
          add_one(a);
9
          printf("a = %d\n", a);
10
          return 0;
11
12
```

```
code/add-one1.c ____
      #include <stdio.h>
                                                                       X
      void add_one(int n)
                                                                       X
3
                                                              stack
                                                                      9050
                                                                           return address
          n = n + 1;
 4
                                                                       5
                                                               base
      int main(void)
                                                                             Χ
      {
 7
          int a = 5;
8
          add_one(a);
9
          printf("a = %d\n", a);
10
          return 0;
11
12
```

```
code/add-one1.c ____
      #include <stdio.h>
                                                                        X
      void add_one(int n)
                                                                        5
                                                                           parameter 1
3
                                                               stack
                                                                      9050
                                                                            return address
          n = n + 1;
                                                                        5
                                                               base
      int main(void)
                                                                              Χ
      {
 7
          int a = 5;
8
          add_one(a);
9
          printf("a = %d\n", a);
10
          return 0;
11
12
```

```
code/add-one1.c ____
                                                             stack
     #include <stdio.h>
     void add_one(int n)
                                                                      5
                                                              base
3
                                                                    9050
          n = n + 1;
4
                                                                      5
     int main(void)
     {
          int a = 5;
8
          add_one(a);
9
          printf("a = %d\n", a);
10
          return 0;
11
12
```

```
code/add-one1.c ____
                                                            stack
     #include <stdio.h>
     void add_one(int n)
                                                                     5
                                                             base
3
                                                                   9050
         n = n + 1;
                                                                     5
     int main(void)
     {
          int a = 5;
8
          add_one(a);
9
          printf("a = %d\n", a);
10
          return 0;
11
12
```

```
code/add-one1.c ____
                                                             stack
     #include <stdio.h>
     void add_one(int n)
                                                                     6
3
                                                             base
                                                                    9050
         n = n + 1:
4
                                                                     5
     int main(void)
6
     {
          int a = 5;
8
          add_one(a);
9
          printf("a = %d\n", a);
10
          return 0;
11
12
```

```
code/add-one1.c ____
     #include <stdio.h>
                                                                     X
     void add_one(int n)
                                                                     6
                                                             stack
3
                                                                    9050
         n = n + 1;
4
                                                                     5
                                                             base
     int main(void)
     {
7
          int a = 5;
8
          add_one(a);
9
          printf("a = %d\n", a);
10
          return 0;
11
12
```

▶ Vi testkör!

```
code/add-one1.c ___
     #include <stdio.h>
                                                                     X
     void add_one(int n)
                                                                     6
                                                             stack
3
                                                                    9050
         n = n + 1;
4
                                                                     5
                                                             base
     int main(void)
     {
7
          int a = 5;
8
          add_one(a);
9
          printf("a = %d\n", a);
10
          return 0;
11
12
```

▶ Vi testkör!

```
code/add-one1.c ___
     #include <stdio.h>
                                                                     X
     void add_one(int n)
                                                                     6
                                                             stack
3
                                                                    9050
         n = n + 1;
4
                                                                     5
                                                             base
     int main(void)
     {
7
          int a = 5;
8
          add_one(a);
9
          printf("a = %d\n", a);
10
          return 0;
11
12
```

▶ Vi testkör!

```
code/add-one1.c _____
     #include <stdio.h>
     void add_one(int n)
3
         n = n + 1;
4
     int main(void)
     {
7
         int a = 5;
8
         add_one(a);
9
         printf("a = %d\n", a);
10
         return 0;
11
12
```

```
X
6
9050
5
```

```
code/add-one2.c
     #include <stdio.h>
     void add_one(int *n)
3
         *n = *n + 1:
     int main(void)
     {
7
         int a = 5;
8
         add_one(&a);
9
         printf("a = %d\n", a);
10
         return 0;
11
12
```

```
312 X
308 X
304 X
300 X
```

```
code/add-one2.c
      #include <stdio.h>
                                                                      X
                                                                 312
      void add_one(int *n)
                                                                      X
                                                                 308
                                                              stack
 3
                                                                      X
                                                                 304
          *n = *n + 1:
 4
                                                                      X
                                                                 300
                                                              base
      int main(void)
      }
 7
          int a = 5;
8
          add_one(&a);
9
          printf("a = %d\n", a);
10
          return 0;
11
12
```

```
code/add-one2.c
      #include <stdio.h>
                                                                      X
                                                                 312
      void add_one(int *n)
                                                                      X
                                                                 308
                                                              stack
3
                                                                      X
                                                                 304
          *n = *n + 1:
 4
                                                                      X
                                                                 300
                                                              base
      int main(void)
 7
          int a = 5;
8
          add_one(&a);
9
          printf("a = %d\n", a);
10
          return 0;
11
12
```

```
code/add-one2.c
      #include <stdio.h>
                                                                      X
                                                                 312
      void add_one(int *n)
                                                                      X
                                                                 308
                                                              stack
3
                                                                      X
                                                                 304
          *n = *n + 1:
 4
                                                                      5
                                                                 300
                                                              base
      int main(void)
      {
 7
          int a = 5;
8
          add_one(&a);
9
          printf("a = %d\n", a);
10
          return 0;
11
12
```

```
code/add-one2.c
      #include <stdio.h>
                                                                        X
                                                                   312
      void add_one(int *n)
                                                                        X
                                                                   308
                                                               stack
 3
                                                                       9050
                                                                            return address
                                                                   304
          *n = *n + 1:
 4
                                                                        5
                                                                   300
                                                                base
      int main(void)
                                                                               Χ
      {
 7
          int a = 5;
8
          add_one(&a);
9
          printf("a = %d\n", a);
10
          return 0;
11
12
```

```
code/add-one2.c
      #include <stdio.h>
                                                                        X
                                                                   312
      void add_one(int *n)
                                                                            parameter 1
                                                                       300
 3
                                                                       9050
                                                                            return address
          *n = *n + 1:
                                                                        5
                                                                base
      int main(void)
                                                                               Χ
      {
 7
          int a = 5;
8
          add_one(&a);
9
          printf("a = %d\n", a);
10
          return 0;
11
12
```

```
code/add-one2.c
                                                             stack
     #include <stdio.h>
                                                                 312
     void add_one(int *n)
                                                                     300
3
                                                                     9050
          *n = *n + 1:
4
                                                                      5
                                                                 300
     int main(void)
     {
          int a = 5;
8
          add_one(&a);
9
          printf("a = %d\n", a);
10
          return 0;
11
12
```

```
code/add-one2.c
                                                           stack
     #include <stdio.h>
     void add_one(int *n)
3
         *n = *n + 1;
     int main(void)
     {
         int a = 5;
8
          add_one(&a);
9
         printf("a = %d\n", a);
10
         return 0;
11
12
```



```
code/add-one2.c
                                                             stack
     #include <stdio.h>
                                                                312
     void add_one(int *n)
                                                                     300
3
                                                                    9050
          *n = *n + 1:
                                                                      6
                                                                300
     int main(void)
6
     {
          int a = 5;
8
          add_one(&a);
9
          printf("a = %d\n", a);
10
          return 0;
11
12
```

```
code/add-one2.c
      #include <stdio.h>
                                                                       X
                                                                  312
      void add_one(int *n)
                                                                      300
                                                                  308
                                                              stack
 3
                                                                      9050
                                                                  304
          *n = *n + 1:
 4
                                                                       6
                                                                  300
                                                               base
      int main(void)
      {
 7
          int a = 5;
8
          add_one(&a);
9
          printf("a = %d\n", a);
10
          return 0;
11
12
```

Vi testkör det andra examplet!

```
code/add-one2.c
      #include <stdio.h>
                                                                       X
                                                                  312
      void add_one(int *n)
                                                                      300
                                                                  308
 3
                                                              stack
                                                                      9050
                                                                  304
          *n = *n + 1:
 4
                                                                       6
                                                                  300
                                                               base
      int main(void)
      {
 7
          int a = 5;
8
          add_one(&a);
9
          printf("a = %d\n", a);
10
          return 0;
11
12
```

Vi testkör det andra examplet!

```
code/add-one2.c
      #include <stdio.h>
                                                                        X
                                                                  312
      void add_one(int *n)
                                                                       300
                                                                  308
                                                               stack `
 3
                                                                      9050
                                                                  304
          *n = *n + 1:
 4
                                                                        6
                                                                  300
                                                                base
      int main(void)
      {
 7
          int a = 5;
8
          add_one(&a);
9
          printf("a = %d\n", a);
10
          return 0;
11
12
```

```
code/add-one2.c
     #include <stdio.h>
     void add_one(int *n)
3
         *n = *n + 1:
4
     int main(void)
     {
7
         int a = 5;
8
          add_one(&a);
9
         printf("a = %d\n", a);
10
         return 0;
11
12
```

```
312 X
308 300
304 9050
300 6
```

```
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:

```
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
 1
      #include <stdio.h>
      void swap(int *v1, int *v2)
 3
 4
          int tmp;
 5
          tmp = *v1;
          *v1 = *v2;
6
          *v2 = tmp;
      int main(void)
9
      {
10
          int n1 = 3, n2 = 418;
11
12
          swap(&n1, &n2);
          printf("n1 = \frac{4.2f}{n}, n2 = \frac{4.2f}{n}, n1, n2);
13
          return 0:
14
15
```

```
324 X
320 X
316 X
312 X
308 X
304 X
300 X
```

Ett till exempel

```
code/swap.c
 1
      #include <stdio.h>
      void swap(int *v1, int *v2)
 3
 4
          int tmp;
 5
          tmp = *v1;
          *v1 = *v2;
6
          *v2 = tmp;
      int main(void)
9
10
          int n1 = 3, n2 = 418;
11
12
          swap(&n1, &n2);
          printf("n1 = \frac{4.2f}{n}, n2 = \frac{4.2f}{n}, n1, n2);
13
          return 0:
14
15
```

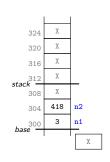
```
X
      324
             X
     320
             X
      316
stack<sup>312</sup>
             X
             X
      308
             X
                   n2
      304
             X
                   n1
 base 300
                       Χ
```

Ett till exempel

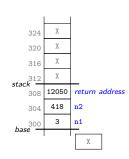
```
code/swap.c
 1
      #include <stdio.h>
      void swap(int *v1, int *v2)
 3
 4
          int tmp;
 5
          tmp = *v1;
          *v1 = *v2;
6
          *v2 = tmp;
      int main(void)
9
10
          int n1 = 3, n2 = 418;
11
12
          swap(&n1, &n2);
          printf("n1 = \frac{4.2f}{n}, n2 = \frac{4.2f}{n}, n1, n2);
13
          return 0:
14
15
```

```
X
      324
              X
     320
              X
      316
stack<sup>312</sup>
              X
              X
      308
              X
                   n2
      304
              X
                   n1
 base 300
                       Χ
```

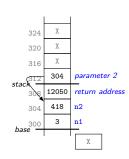
```
code/swap.c
 1
      #include <stdio.h>
      void swap(int *v1, int *v2)
 3
 4
          int tmp;
 5
          tmp = *v1;
          *v1 = *v2;
6
          *v2 = tmp;
      int main(void)
9
      {
10
          int n1 = 3, n2 = 418;
11
          swap(&n1, &n2);
12
          printf("n1 = \frac{4.2f}{n^2}, n2 = \frac{4.2f}{n}", n1, n2);
13
          return 0:
14
15
```



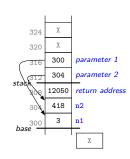
```
code/swap.c
 1
      #include <stdio.h>
      void swap(int *v1, int *v2)
 3
 4
          int tmp;
 5
          tmp = *v1;
          *v1 = *v2:
6
          *v2 = tmp;
      int main(void)
9
      {
10
          int n1 = 3, n2 = 418;
11
12
          swap(&n1, &n2);
          printf("n1 = \frac{4.2f}{n2}, n2 = \frac{4.2f}{n}, n1, n2);
13
          return 0:
14
15
```



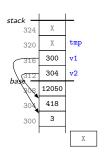
```
code/swap.c
 1
      #include <stdio.h>
      void swap(int *v1, int *v2)
 3
 4
          int tmp;
 5
          tmp = *v1;
          *v1 = *v2:
6
          *v2 = tmp;
      int main(void)
9
      {
10
          int n1 = 3, n2 = 418;
11
12
          swap(&n1, &n2);
          printf("n1 = \frac{4.2f}{n2}, n2 = \frac{4.2f}{n}, n1, n2);
13
          return 0:
14
15
```



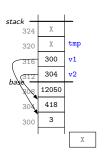
```
code/swap.c
 1
      #include <stdio.h>
      void swap(int *v1, int *v2)
 3
 4
          int tmp;
 5
          tmp = *v1;
          *v1 = *v2:
6
          *v2 = tmp;
      int main(void)
9
      {
10
          int n1 = 3, n2 = 418;
11
12
          swap(&n1, &n2);
          printf("n1 = \frac{4.2f}{n^2}, n2 = \frac{4.2f}{n}", n1, n2);
13
          return 0:
14
15
```



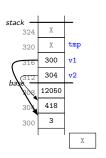
```
code/swap.c
 1
      #include <stdio.h>
      void swap(int *v1, int *v2)
 3
          int tmp;
 4
 5
          tmp = *v1;
          *v1 = *v2;
6
          *v2 = tmp;
      int main(void)
9
      {
10
          int n1 = 3, n2 = 418;
11
12
          swap(&n1, &n2);
          printf("n1 = \frac{4.2f}{n^2}, n2 = \frac{4.2f}{n}", n1, n2);
13
          return 0:
14
15
```



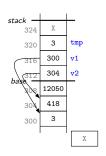
```
code/swap.c
 1
      #include <stdio.h>
      void swap(int *v1, int *v2)
 3
 4
          int tmp;
 5
          tmp = *v1;
          *v1 = *v2:
6
          *v2 = tmp;
      int main(void)
      {
10
          int n1 = 3, n2 = 418;
11
12
          swap(&n1, &n2);
          printf("n1 = \frac{4.2f}{n^2}, n2 = \frac{4.2f}{n}", n1, n2);
13
          return 0:
14
15
```



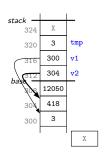
```
code/swap.c
 1
      #include <stdio.h>
      void swap(int *v1, int *v2)
 3
 4
          int tmp;
 5
          tmp = *v1;
          *v1 = *v2:
6
          *v2 = tmp;
      int main(void)
      {
10
          int n1 = 3, n2 = 418;
11
12
          swap(&n1, &n2);
          printf("n1 = \frac{4.2f}{n^2}, n2 = \frac{4.2f}{n}", n1, n2);
13
          return 0:
14
15
```



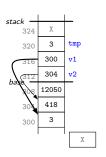
```
code/swap.c
 1
      #include <stdio.h>
      void swap(int *v1, int *v2)
 3
 4
          int tmp;
 5
          tmp = *v1;
          *v1 = *v2;
6
          *v2 = tmp;
      int main(void)
      {
10
          int n1 = 3, n2 = 418;
11
12
          swap(&n1, &n2);
          printf("n1 = \frac{4.2f}{n^2}, n2 = \frac{4.2f}{n}", n1, n2);
13
          return 0:
14
15
```



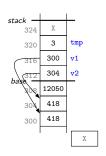
```
code/swap.c
 1
      #include <stdio.h>
      void swap(int *v1, int *v2)
 3
 4
          int tmp;
 5
          tmp = *v1;
          *v1 = *v2;
6
          *v2 = tmp;
      int main(void)
      {
10
          int n1 = 3, n2 = 418;
11
12
          swap(&n1, &n2);
          printf("n1 = \frac{4.2f}{n^2}, n2 = \frac{4.2f}{n}", n1, n2);
13
          return 0:
14
15
```



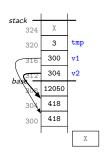
```
code/swap.c
 1
      #include <stdio.h>
      void swap(int *v1, int *v2)
 3
 4
          int tmp;
 5
          tmp = *v1;
          *v1 = *v2;
6
          *v2 = tmp;
      int main(void)
      {
10
          int n1 = 3, n2 = 418;
11
12
          swap(&n1, &n2);
          printf("n1 = \frac{4.2f}{n^2}, n2 = \frac{4.2f}{n}", n1, n2);
13
          return 0:
14
15
```



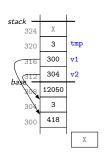
```
code/swap.c
 1
      #include <stdio.h>
      void swap(int *v1, int *v2)
 3
 4
          int tmp;
 5
          tmp = *v1;
          *v1 = *v2;
6
          *v2 = tmp;
      int main(void)
9
      {
10
          int n1 = 3, n2 = 418;
11
12
          swap(&n1, &n2);
          printf("n1 = \frac{4.2f}{n^2}, n2 = \frac{4.2f}{n}", n1, n2);
13
          return 0:
14
15
```



```
code/swap.c
 1
      #include <stdio.h>
      void swap(int *v1, int *v2)
 3
 4
          int tmp;
 5
          tmp = *v1;
          *v1 = *v2;
6
          *v2 = tmp;
      int main(void)
9
      {
10
          int n1 = 3, n2 = 418;
11
12
          swap(&n1, &n2);
          printf("n1 = \frac{4.2f}{n^2}, n2 = \frac{4.2f}{n}", n1, n2);
13
          return 0:
14
15
```



```
code/swap.c
 1
      #include <stdio.h>
      void swap(int *v1, int *v2)
 3
 4
          int tmp;
 5
          tmp = *v1;
          *v1 = *v2:
6
          *v2 = tmp;
      int main(void)
      {
10
          int n1 = 3, n2 = 418;
11
12
          swap(&n1, &n2);
          printf("n1 = \frac{4.2f}{n^2}, n2 = \frac{4.2f}{n}", n1, n2);
13
          return 0:
14
15
```



```
code/swap.c
 1
      #include <stdio.h>
      void swap(int *v1, int *v2)
 3
 4
          int tmp;
 5
          tmp = *v1;
          *v1 = *v2;
6
          *v2 = tmp;
      int main(void)
9
      {
10
          int n1 = 3, n2 = 418;
11
12
          swap(&n1, &n2);
          printf("n1 = \frac{4.2f}{n2}, n2 = \frac{4.2f}{n}, n1, n2);
13
14
          return 0:
15
```

```
X
     324
           3
    320
          300
     316
stack 312
          304
         12050
     308
           3
                n2
     304
          418
                n1
 base 300
                   Χ
```

```
code/swap.c
 1
      #include <stdio.h>
      void swap(int *v1, int *v2)
 3
 4
          int tmp;
 5
          tmp = *v1;
          *v1 = *v2:
6
          *v2 = tmp;
      int main(void)
9
      {
10
          int n1 = 3, n2 = 418;
11
12
          swap(&n1, &n2);
          printf("n1 = \frac{4.2f}{n2} = \frac{4.2f}{n}", n1, n2);
13
          return 0:
14
15
```

```
X
     324
           3
    320
          300
     316
stack 312
          304
         12050
     308
           3
                n2
     304
          418
                n1
 base 300
                   Χ
```

n1 = 418, n2 = 3

```
code/swap.c
 1
      #include <stdio.h>
      void swap(int *v1, int *v2)
 3
 4
          int tmp;
 5
          tmp = *v1;
          *v1 = *v2:
6
          *v2 = tmp;
      int main(void)
9
      {
10
          int n1 = 3, n2 = 418;
11
12
          swap(&n1, &n2);
          printf("n1 = \frac{4.2f}{n^2}, n2 = \frac{4.2f}{n}", n1, n2);
13
          return 0:
14
15
```

```
X
     324
           3
    320
          300
     316
stack 312
          304
         12050
     308
           3
                n2
     304
          418
                n1
 base 300
                    0
```

n1 = 418, n2 = 3

```
code/swap.c
 1
      #include <stdio.h>
      void swap(int *v1, int *v2)
 3
 4
          int tmp;
 5
          tmp = *v1;
          *v1 = *v2:
6
          *v2 = tmp;
      int main(void)
9
      {
10
          int n1 = 3, n2 = 418;
11
12
          swap(&n1, &n2);
          printf("n1 = \frac{4.2f}{n^2}, n2 = \frac{4.2f}{n}", n1, n2);
13
          return 0:
14
15
```

```
324 X
320 3
316 300
312 304
308 12050
304 3
300 418
```

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)

```
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

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

- Exempel med noder som lokala variabler i main (struct-on-stack)
- Exempel med noder med typat värde som skapas dynamiskt (dynamic-struct)
- 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 _____
     typedef struct node {
         int val; // typed payload of fixed size inside struct
         struct node *next;
4
     } node;
5
                           code/struct-on-stack.c _
     void print list(const node *p)
6
     {
         printf("( ");
         while (p != NULL) {
9
10
             printf("%d ", p->val); // we know the type of val
             p = p->next;
11
12
         printf(")\n");
13
14
```

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

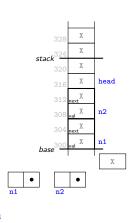
```
code/struct-on-stack.c
     int main(void)
15
16
         node n1; // automatic allocation on stack
17
          node n2;
18
         node *head;
19
         n2.val = 21;
20
         n2.next = NULL;
21
         n1.val = 20;
22
23
         n1.next = &n2:
24
         head = &n1:
         print_list(head);
25
          return 0:
26
     } // automatic cleanup
27
```

► Allokering och avallokering kommer att ske automatiskt

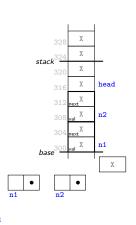
```
#include <stdio.h>
     typedef struct node {
 3
         int val; // typed payload of fixed size inside struct
 4
         struct node *next;
    } node;
     void print_list(const node *p)
 8
         printf("( ");
 9
         while (p != NULL) {
10
             printf("%d ", p->val); // we know the type of val
11
             p = p->next;
12
13
         printf(")\n"):
14
15
     int main(void)
16
17
         node n1; // automatic allocation on stack
18
         node n2;
19
         node *head:
20
         n2.val = 21:
21
         n2.next = NULL;
22
         n1.val = 20:
23
         n1.next = &n2:
24
         head = &n1;
25
         print_list(head);
26
         return 0:
27
    } // automatic cleanup
```



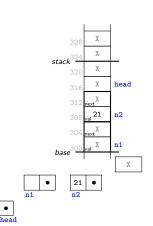
```
#include <stdio.h>
     typedef struct node {
 3
         int val; // typed payload of fixed size inside struct
 4
         struct node *next;
     } node;
     void print_list(const node *p)
 8
         printf("( ");
 9
         while (p != NULL) {
10
             printf("%d ", p->val); // we know the type of val
11
             p = p->next;
12
13
         printf(")\n"):
14
15
     int main(void)
16
17
         node n1; // automatic allocation on stack
18
         node n2;
19
         node *head:
20
         n2.val = 21:
21
         n2.next = NULL;
22
         n1.val = 20:
23
         n1.next = &n2:
24
         head = &n1;
25
         print_list(head);
26
         return 0:
27
     } // automatic cleanup
```



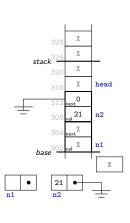
```
#include <stdio.h>
     typedef struct node {
 3
         int val; // typed payload of fixed size inside struct
 4
         struct node *next;
    } node;
     void print_list(const node *p)
 8
         printf("( ");
 9
         while (p != NULL) {
10
             printf("%d ", p->val); // we know the type of val
11
             p = p->next;
12
13
         printf(")\n"):
14
15
     int main(void)
16
17
         node n1; // automatic allocation on stack
18
         node n2;
19
         node *head:
20
         n2.val = 21:
21
         n2.next = NULL;
22
         n1.val = 20:
23
         n1.next = &n2:
24
         head = &n1;
25
         print_list(head);
26
         return 0:
27
     } // automatic cleanup
```



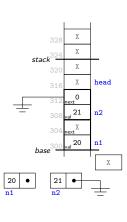
```
#include <stdio.h>
     typedef struct node {
 3
         int val; // typed payload of fixed size inside struct
 4
         struct node *next;
    } node;
     void print_list(const node *p)
 8
         printf("( ");
 9
         while (p != NULL) {
10
             printf("%d ", p->val); // we know the type of val
11
             p = p->next;
12
13
         printf(")\n"):
14
15
     int main(void)
16
17
         node n1; // automatic allocation on stack
18
         node n2;
19
         node *head:
20
         n2.val = 21:
21
         n2.next = NULL;
22
         n1.val = 20:
23
         n1.next = &n2:
24
         head = &n1;
25
         print_list(head);
26
         return 0:
27
     } // automatic cleanup
```



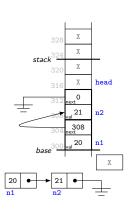
```
#include <stdio.h>
     typedef struct node {
 3
         int val; // typed payload of fixed size inside struct
 4
         struct node *next;
     } node;
     void print_list(const node *p)
 8
         printf("( ");
 9
         while (p != NULL) {
10
             printf("%d ", p->val); // we know the type of val
11
             p = p->next;
12
13
         printf(")\n"):
14
15
     int main(void)
16
17
         node n1; // automatic allocation on stack
18
         node n2;
19
         node *head:
20
         n2.val = 21:
21
         n2.next = NULL;
22
         n1.val = 20:
23
         n1.next = &n2:
24
         head = &n1;
25
         print_list(head);
26
         return 0:
27
     } // automatic cleanup
```



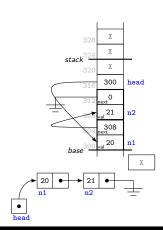
```
#include <stdio.h>
     typedef struct node {
 3
         int val; // typed payload of fixed size inside struct
 4
         struct node *next;
     } node;
     void print_list(const node *p)
 8
         printf("( ");
 9
         while (p != NULL) {
10
             printf("%d ", p->val); // we know the type of val
11
             p = p->next;
12
13
         printf(")\n"):
14
15
     int main(void)
16
17
         node n1; // automatic allocation on stack
18
         node n2;
19
         node *head:
20
         n2.val = 21:
21
         n2.next = NULL;
22
         n1.val = 20:
23
         n1.next = &n2:
24
         head = &n1;
25
         print_list(head);
26
         return 0:
27
     } // automatic cleanup
```



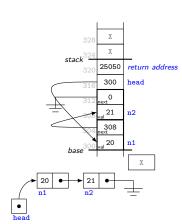
```
#include <stdio.h>
     typedef struct node {
 3
         int val; // typed payload of fixed size inside struct
 4
         struct node *next;
     } node;
     void print_list(const node *p)
 8
         printf("( ");
 9
         while (p != NULL) {
10
             printf("%d ", p->val); // we know the type of val
11
             p = p->next;
12
13
         printf(")\n"):
14
15
     int main(void)
16
17
         node n1; // automatic allocation on stack
18
         node n2;
19
         node *head:
20
         n2.val = 21:
21
         n2.next = NULL;
22
         n1.val = 20:
23
         n1.next = &n2:
24
         head = &n1;
25
         print_list(head);
26
         return 0:
27
     } // automatic cleanup
```



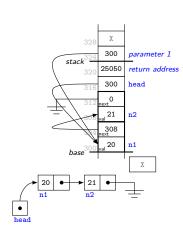
```
#include <stdio.h>
     typedef struct node {
         int val; // typed payload of fixed size inside struct
         struct node *next;
 4
    } node;
     void print_list(const node *p)
 8
         printf("( ");
 9
         while (p != NULL) {
10
             printf("%d ", p->val); // we know the type of val
11
             p = p->next;
12
13
         printf(")\n"):
14
15
     int main(void)
16
17
         node n1; // automatic allocation on stack
18
         node n2;
19
         node *head:
20
         n2.val = 21:
21
         n2.next = NULL;
22
         n1.val = 20:
23
         n1.next = &n2:
24
         head = &n1;
25
         print_list(head);
26
         return 0:
27
     } // automatic cleanup
```



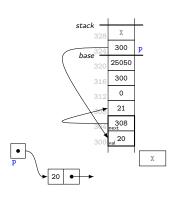
```
#include <stdio.h>
     typedef struct node {
         int val; // typed payload of fixed size inside struct
 4
         struct node *next;
    } node;
     void print_list(const node *p)
 8
         printf("( ");
 9
         while (p != NULL) {
10
             printf("%d ", p->val); // we know the type of val
11
             p = p->next;
12
13
         printf(")\n"):
14
15
     int main(void)
16
17
         node n1; // automatic allocation on stack
18
         node n2;
19
         node *head:
20
         n2.val = 21:
21
         n2.next = NULL;
22
         n1.val = 20:
23
         n1.next = &n2:
24
         head = &n1;
25
         print_list(head);
26
         return 0:
27
     } // automatic cleanup
```



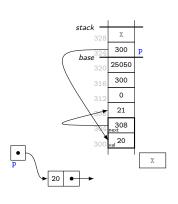
```
#include <stdio.h>
     typedef struct node {
         int val; // typed payload of fixed size inside struct
 4
         struct node *next;
    } node;
     void print_list(const node *p)
 8
         printf("( ");
 9
         while (p != NULL) {
10
             printf("%d ", p->val); // we know the type of val
11
             p = p->next;
12
13
         printf(")\n"):
14
15
     int main(void)
16
17
         node n1; // automatic allocation on stack
18
         node n2;
19
         node *head:
20
         n2.val = 21:
21
         n2.next = NULL;
22
         n1.val = 20:
23
         n1.next = &n2:
24
         head = &n1;
25
         print_list(head);
26
         return 0:
27
     } // automatic cleanup
```



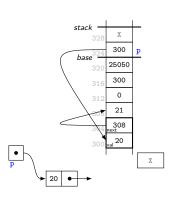
```
#include <stdio.h>
     typedef struct node {
 3
         int val; // typed payload of fixed size inside struct
 4
         struct node *next;
     } node;
     void print_list(const node *p)
 8
         printf("( ");
 9
         while (p != NULL) {
10
             printf("%d ", p->val); // we know the type of val
11
             p = p->next;
12
13
         printf(")\n"):
14
15
     int main(void)
16
17
         node n1; // automatic allocation on stack
18
         node n2;
19
         node *head:
20
         n2.val = 21:
21
         n2.next = NULL;
22
         n1.val = 20:
23
         n1.next = &n2:
24
         head = &n1;
25
         print_list(head);
26
         return 0:
27
     } // automatic cleanup
```



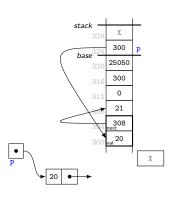
```
#include <stdio.h>
     typedef struct node {
 3
         int val; // typed payload of fixed size inside struct
         struct node *next;
 4
     } node;
     void print_list(const node *p)
 8
         printf("( ");
 9
         while (p != NULL) {
10
             printf("%d ", p->val); // we know the type of val
11
             p = p->next;
12
13
         printf(")\n"):
14
15
     int main(void)
16
17
         node n1; // automatic allocation on stack
18
         node n2;
19
         node *head:
20
         n2.val = 21:
21
         n2.next = NULL;
22
         n1.val = 20:
23
         n1.next = &n2:
24
         head = &n1;
25
         print_list(head);
26
         return 0:
27
     } // automatic cleanup
```



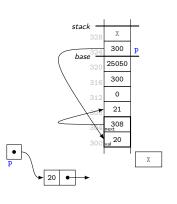
```
#include <stdio.h>
     typedef struct node {
         int val; // typed payload of fixed size inside struct
 4
         struct node *next;
     } node;
     void print_list(const node *p)
 8
         printf("( ");
         while (p != NULL) {
 9
10
             printf("%d ", p->val); // we know the type of val
11
             p = p->next;
12
13
         printf(")\n"):
14
15
     int main(void)
16
17
         node n1; // automatic allocation on stack
18
         node n2;
19
         node *head:
20
         n2.val = 21:
21
         n2.next = NULL;
22
         n1.val = 20:
23
         n1.next = &n2:
24
         head = &n1;
25
         print_list(head);
26
         return 0:
27
     } // automatic cleanup
```



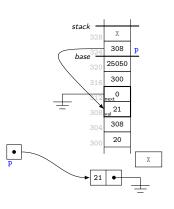
```
#include <stdio.h>
     typedef struct node {
 3
         int val; // typed payload of fixed size inside struct
         struct node *next;
 4
     } node;
     void print_list(const node *p)
 8
         printf("( ");
         while (p != NULL) {
 9
10
             printf("%d ", p->val); // we know the type of val
11
             p = p->next;
12
13
         printf(")\n"):
14
15
     int main(void)
16
17
         node n1; // automatic allocation on stack
18
         node n2;
19
         node *head:
20
         n2.val = 21:
21
         n2.next = NULL;
22
         n1.val = 20:
23
         n1.next = &n2:
24
         head = &n1;
25
         print_list(head);
26
         return 0:
27
     } // automatic cleanup
```



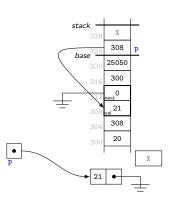
```
#include <stdio.h>
     typedef struct node {
         int val; // typed payload of fixed size inside struct
 4
         struct node *next;
     } node;
     void print_list(const node *p)
 8
         printf("( ");
 9
         while (p != NULL) {
10
             printf("%d ", p->val); // we know the type of val
11
             p = p->next;
12
13
         printf(")\n"):
14
15
     int main(void)
16
17
         node n1; // automatic allocation on stack
18
         node n2;
19
         node *head:
20
         n2.val = 21:
21
         n2.next = NULL;
22
         n1.val = 20:
23
         n1.next = &n2:
24
         head = &n1;
25
         print_list(head);
26
         return 0:
27
     } // automatic cleanup
```



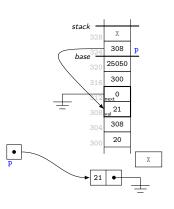
```
#include <stdio.h>
     typedef struct node {
 3
         int val; // typed payload of fixed size inside struct
 4
         struct node *next;
     } node;
     void print_list(const node *p)
 8
         printf("( ");
         while (p != NULL) {
 9
10
             printf("%d ", p->val); // we know the type of val
11
             p = p->next;
12
13
         printf(")\n"):
14
15
     int main(void)
16
17
         node n1; // automatic allocation on stack
18
         node n2;
19
         node *head:
20
         n2.val = 21:
21
         n2.next = NULL;
22
         n1.val = 20:
23
         n1.next = &n2:
24
         head = &n1;
25
         print_list(head);
26
         return 0:
27
     } // automatic cleanup
```



```
#include <stdio.h>
     typedef struct node {
 3
         int val; // typed payload of fixed size inside struct
         struct node *next;
 4
     } node;
     void print_list(const node *p)
 8
         printf("( ");
         while (p != NULL) {
 9
10
             printf("%d ", p->val); // we know the type of val
11
             p = p->next;
12
13
         printf(")\n"):
14
15
     int main(void)
16
17
         node n1; // automatic allocation on stack
18
         node n2;
19
         node *head:
20
         n2.val = 21:
21
         n2.next = NULL;
22
         n1.val = 20:
23
         n1.next = &n2:
24
         head = &n1;
25
         print_list(head);
26
         return 0:
27
     } // automatic cleanup
```

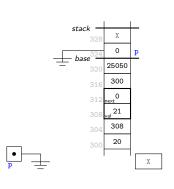


```
#include <stdio.h>
     typedef struct node {
 3
         int val; // typed payload of fixed size inside struct
 4
         struct node *next;
     } node;
     void print_list(const node *p)
 8
         printf("( ");
 9
         while (p != NULL) {
10
             printf("%d ", p->val); // we know the type of val
11
             p = p->next;
12
13
         printf(")\n"):
14
15
     int main(void)
16
17
         node n1; // automatic allocation on stack
18
         node n2;
19
         node *head:
20
         n2.val = 21:
21
         n2.next = NULL;
22
         n1.val = 20:
23
         n1.next = &n2:
24
         head = &n1;
25
         print_list(head);
26
         return 0:
27
     } // automatic cleanup
```



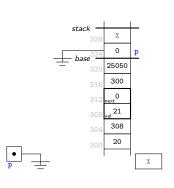
(20 21

```
#include <stdio.h>
     typedef struct node {
 3
         int val; // typed payload of fixed size inside struct
 4
         struct node *next;
     } node;
     void print_list(const node *p)
 8
         printf("( ");
 9
         while (p != NULL) {
10
             printf("%d ", p->val); // we know the type of val
11
             p = p->next;
12
13
         printf(")\n"):
14
15
     int main(void)
16
17
         node n1; // automatic allocation on stack
18
         node n2;
19
         node *head:
20
         n2.val = 21:
21
         n2.next = NULL;
22
         n1.val = 20:
23
         n1.next = &n2:
24
         head = &n1;
25
         print_list(head);
26
         return 0:
27
     } // automatic cleanup
```



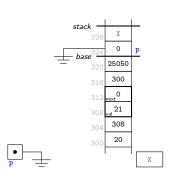
(20 21

```
#include <stdio.h>
     typedef struct node {
 3
         int val; // typed payload of fixed size inside struct
 4
         struct node *next;
     } node;
     void print_list(const node *p)
 8
         printf("( ");
 9
         while (p != NULL) {
10
             printf("%d ", p->val); // we know the type of val
11
             p = p->next;
12
13
         printf(")\n");
14
15
     int main(void)
16
17
         node n1; // automatic allocation on stack
18
         node n2;
19
         node *head:
20
         n2.val = 21:
21
         n2.next = NULL;
22
         n1.val = 20:
23
         n1.next = &n2:
24
         head = &n1;
25
         print_list(head);
26
         return 0:
27
     } // automatic cleanup
```

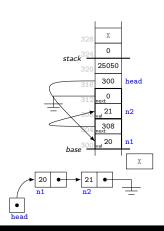


(20 21

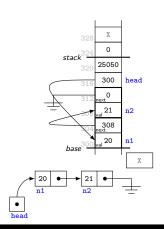
```
#include <stdio.h>
     typedef struct node {
 3
         int val; // typed payload of fixed size inside struct
 4
         struct node *next;
     } node;
     void print_list(const node *p)
 8
         printf("( ");
 9
         while (p != NULL) {
10
             printf("%d ", p->val); // we know the type of val
11
             p = p->next;
12
13
         printf(")\n");
14
15
     int main(void)
16
17
         node n1; // automatic allocation on stack
18
         node n2;
19
         node *head:
20
         n2.val = 21:
21
         n2.next = NULL;
22
         n1.val = 20:
23
         n1.next = &n2:
24
         head = &n1;
25
         print_list(head);
26
         return 0:
27
     } // automatic cleanup
```



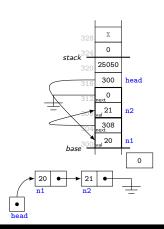
```
#include <stdio.h>
     typedef struct node {
         int val; // typed payload of fixed size inside struct
 4
         struct node *next;
    } node;
     void print_list(const node *p)
 8
         printf("( ");
 9
         while (p != NULL) {
10
             printf("%d ", p->val); // we know the type of val
11
             p = p->next;
12
13
         printf(")\n"):
14
15
     int main(void)
16
17
         node n1; // automatic allocation on stack
18
         node n2;
19
         node *head:
20
         n2.val = 21:
21
         n2.next = NULL;
22
         n1.val = 20:
23
         n1.next = &n2:
24
         head = &n1;
25
         print_list(head);
26
         return 0:
27
     } // automatic cleanup
```



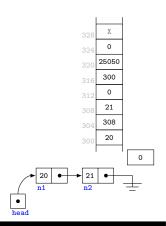
```
#include <stdio.h>
     typedef struct node {
         int val; // typed payload of fixed size inside struct
 4
         struct node *next;
    } node;
     void print_list(const node *p)
 8
         printf("( ");
 9
         while (p != NULL) {
10
             printf("%d ", p->val); // we know the type of val
11
             p = p->next;
12
13
         printf(")\n"):
14
15
     int main(void)
16
17
         node n1; // automatic allocation on stack
18
         node n2;
19
         node *head:
20
         n2.val = 21:
21
         n2.next = NULL;
22
         n1.val = 20:
23
         n1.next = &n2:
24
         head = &n1;
25
         print_list(head);
26
         return 0:
27
     } // automatic cleanup
```



```
#include <stdio.h>
     typedef struct node {
         int val; // typed payload of fixed size inside struct
 4
         struct node *next;
    } node;
     void print_list(const node *p)
 8
         printf("( ");
 9
         while (p != NULL) {
10
             printf("%d ", p->val); // we know the type of val
11
             p = p->next;
12
13
         printf(")\n"):
14
15
     int main(void)
16
17
         node n1; // automatic allocation on stack
18
         node n2;
19
         node *head:
20
         n2.val = 21:
21
         n2.next = NULL;
22
         n1.val = 20:
23
         n1.next = &n2:
24
         head = &n1;
25
         print_list(head);
26
         return 0:
27
      // automatic cleanup
```



```
#include <stdio.h>
     typedef struct node {
 3
         int val; // typed payload of fixed size inside struct
 4
         struct node *next;
    } node;
     void print_list(const node *p)
 8
         printf("( ");
 9
         while (p != NULL) {
10
             printf("%d ", p->val); // we know the type of val
11
             p = p->next;
12
13
         printf(")\n"):
14
15
     int main(void)
16
17
         node n1; // automatic allocation on stack
18
         node n2;
19
         node *head:
20
         n2.val = 21:
21
         n2.next = NULL;
22
         n1.val = 20:
23
         n1.next = &n2:
24
         head = &n1;
25
         print_list(head);
26
         return 0:
27
    } // automatic cleanup
```



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
 - 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:

```
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

```
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

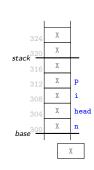
▶ I lista 2 allokeras och avallokeras noderna dynamiskt

```
code/dynamic-struct.c _____
     int main(void)
16
17
          node *n, *head = NULL;
18
          for (int i = 0: i < 2: i++) {
19
              n = malloc(sizeof(*n));
20
              n->val = 21 - i;
21
22
              n->next = head:
              head = n;
23
24
          print_list(head);
25
26
         n = head; // Explicit cleanup
          while (n != NULL) {
27
28
              node *p = n->next;
              free(n);
29
30
              n = p;
31
          return 0;
32
33
```

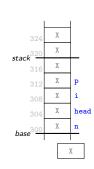
```
typedef struct node {
 4
         int val: // tuped pauload of fixed size inside struct
 5
         struct node *next:
 6
    } node;
     void print_list(const node *p)
 8
 9
         printf("( ");
10
         while (p != NULL) {
11
             printf("%d ", p->val); // we know the type of val
12
             p = p->next:
13
14
         printf(")\n"):
15
16
     int main(void)
17
18
         node *n. *head = NULL:
19
         for (int i = 0; i < 2; i++) {
20
             n = malloc(sizeof(*n));
21
            n->val = 21 - i:
22
             n->next = head:
23
             head = n;
24
25
         print list(head):
26
         n = head; // Explicit cleanup
27
         while (n != NULL) {
28
             node *p = n->next;
29
             free(n);
30
             n = p;
31
32
         return 0;
33
```



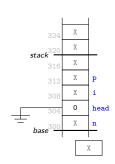
```
typedef struct node {
 4
         int val: // tuped pauload of fixed size inside struct
 5
         struct node *next:
 6
    } node;
     void print_list(const node *p)
 8
 9
         printf("( ");
10
         while (p != NULL) {
11
             printf("%d ", p->val); // we know the type of val
12
             p = p->next:
13
14
         printf(")\n"):
15
16
     int main(void)
17
18
         node *n. *head = NULL:
19
         for (int i = 0; i < 2; i++) {
20
             n = malloc(sizeof(*n));
21
             n->val = 21 - i:
22
             n->next = head:
23
             head = n;
24
25
         print list(head):
26
         n = head; // Explicit cleanup
27
         while (n != NULL) {
28
             node *p = n->next:
29
             free(n);
30
             n = p;
31
32
         return 0;
33
```



```
typedef struct node {
 4
         int val: // tuped pauload of fixed size inside struct
 5
         struct node *next:
 6
    } node;
     void print_list(const node *p)
 8
 9
         printf("( ");
10
         while (p != NULL) {
11
             printf("%d ", p->val); // we know the type of val
12
             p = p->next:
13
14
         printf(")\n"):
15
16
     int main(void)
17
18
         node *n. *head = NULL:
19
         for (int i = 0; i < 2; i++) {
20
             n = malloc(sizeof(*n));
21
             n->val = 21 - i:
22
             n->next = head:
23
             head = n;
24
25
         print list(head):
26
         n = head; // Explicit cleanup
27
         while (n != NULL) {
28
             node *p = n->next:
29
             free(n);
30
             n = p;
31
32
         return 0;
33
```

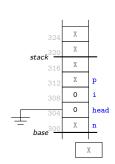


```
typedef struct node {
 4
         int val: // tuped pauload of fixed size inside struct
 5
         struct node *next:
 6
     } node;
     void print_list(const node *p)
 8
 9
         printf("( ");
10
         while (p != NULL) {
11
             printf("%d ", p->val); // we know the type of val
12
             p = p->next:
13
14
         printf(")\n"):
15
16
     int main(void)
17
18
         node *n. *head = NULL:
19
         for (int i = 0; i < 2; i++) {
20
             n = malloc(sizeof(*n));
21
             n->val = 21 - i:
22
             n->next = head:
23
             head = n;
24
25
         print list(head):
26
         n = head; // Explicit cleanup
27
         while (n != NULL) {
28
             node *p = n->next;
29
             free(n);
30
             n = p;
31
32
         return 0;
33
```



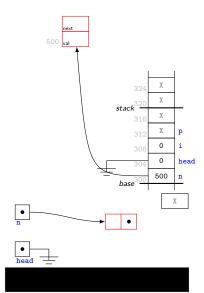


```
typedef struct node {
 4
         int val: // tuped pauload of fixed size inside struct
 5
         struct node *next:
 6
     } node;
     void print_list(const node *p)
 8
 9
         printf("( ");
10
         while (p != NULL) {
11
             printf("%d ", p->val); // we know the type of val
12
             p = p->next:
13
14
         printf(")\n"):
15
16
     int main(void)
17
18
         node *n. *head = NULL:
19
         for (int i = 0; i < 2; i++) {
20
             n = malloc(sizeof(*n));
21
             n->val = 21 - i:
22
             n->next = head:
23
             head = n;
24
25
         print list(head):
26
         n = head; // Explicit cleanup
27
         while (n != NULL) {
28
             node *p = n->next;
29
             free(n);
30
             n = p;
31
32
         return 0;
33
```

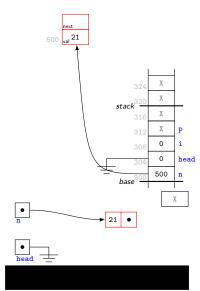




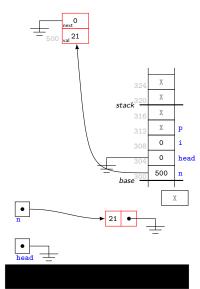
```
typedef struct node {
 4
         int val: // tuped pauload of fixed size inside struct
 5
         struct node *next:
 6
     } node;
     void print_list(const node *p)
 8
 9
         printf("( ");
10
         while (p != NULL) {
11
             printf("%d ", p->val); // we know the type of val
12
             p = p->next:
13
14
         printf(")\n"):
15
16
     int main(void)
17
18
         node *n. *head = NULL:
19
         for (int i = 0; i < 2; i++) {
20
             n = malloc(sizeof(*n));
21
             n->val = 21 - i:
22
             n->next = head:
23
             head = n;
24
25
         print list(head):
26
         n = head; // Explicit cleanup
27
         while (n != NULL) {
28
             node *p = n->next;
29
             free(n);
30
             n = p;
31
32
         return 0;
33
```



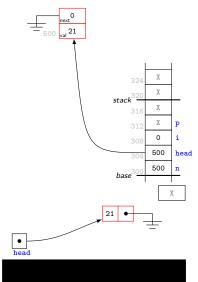
```
typedef struct node {
 4
         int val: // tuped pauload of fixed size inside struct
 5
         struct node *next:
 6
     } node;
     void print_list(const node *p)
 8
 9
         printf("( ");
10
         while (p != NULL) {
11
             printf("%d ", p->val); // we know the type of val
12
             p = p->next:
13
14
         printf(")\n"):
15
16
     int main(void)
17
18
         node *n. *head = NULL:
19
         for (int i = 0; i < 2; i++) {
20
             n = malloc(sizeof(*n));
21
             n->val = 21 - i:
22
             n->next = head:
23
             head = n;
24
25
         print list(head):
26
         n = head; // Explicit cleanup
27
         while (n != NULL) {
28
             node *p = n->next;
29
             free(n);
30
             n = p;
31
32
         return 0;
33
```



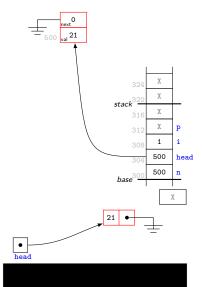
```
typedef struct node {
 4
         int val: // tuped pauload of fixed size inside struct
 5
         struct node *next:
 6
     } node;
     void print_list(const node *p)
 8
 9
         printf("( ");
10
         while (p != NULL) {
11
             printf("%d ", p->val); // we know the type of val
12
             p = p->next:
13
14
         printf(")\n"):
15
16
     int main(void)
17
18
         node *n. *head = NULL:
19
         for (int i = 0; i < 2; i++) {
20
             n = malloc(sizeof(*n));
21
             n->val = 21 - i:
22
             n->next = head:
23
             head = n;
24
25
         print list(head):
26
         n = head; // Explicit cleanup
27
         while (n != NULL) {
28
             node *p = n->next;
29
             free(n);
30
             n = p;
31
32
         return 0;
33
```



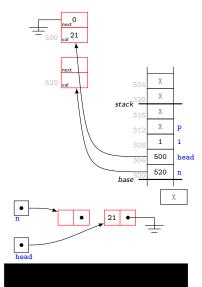
```
typedef struct node {
 4
         int val: // tuped pauload of fixed size inside struct
 5
         struct node *next:
 6
    } node;
     void print_list(const node *p)
 8
 9
         printf("( ");
10
         while (p != NULL) {
11
             printf("%d ", p->val); // we know the type of val
12
             p = p->next:
13
         printf(")\n"):
14
15
16
     int main(void)
17
18
         node *n. *head = NULL:
19
         for (int i = 0; i < 2; i++) {
20
             n = malloc(sizeof(*n));
21
             n->val = 21 - i:
22
             n->next = head:
23
             head = n;
24
25
         print list(head):
26
         n = head; // Explicit cleanup
27
         while (n != NULL) {
28
             node *p = n->next:
29
             free(n);
30
             n = p;
31
32
         return 0;
33
```



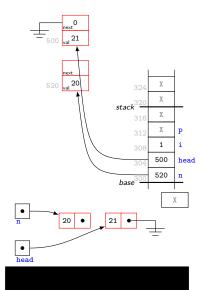
```
typedef struct node {
 4
         int val: // tuped pauload of fixed size inside struct
 5
         struct node *next:
 6
    } node;
     void print_list(const node *p)
 8
 9
         printf("( ");
10
         while (p != NULL) {
11
             printf("%d ", p->val); // we know the type of val
12
             p = p->next:
13
         printf(")\n"):
14
15
16
     int main(void)
17
18
         node *n. *head = NULL:
19
         for (int i = 0; i < 2; i++) {
20
             n = malloc(sizeof(*n));
21
             n->val = 21 - i:
22
             n->next = head:
23
             head = n;
24
25
         print list(head):
26
         n = head; // Explicit cleanup
27
         while (n != NULL) {
28
             node *p = n->next;
29
             free(n);
30
             n = p;
31
32
         return 0;
33
```



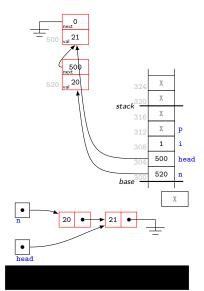
```
typedef struct node {
         int val; // typed payload of fixed size inside struct
 4
 5
         struct node *next:
 6
    } node;
     void print_list(const node *p)
 8
 9
         printf("( ");
10
         while (p != NULL) {
11
             printf("%d ", p->val); // we know the type of val
12
             p = p->next:
13
         printf(")\n"):
14
15
16
     int main(void)
17
18
         node *n. *head = NULL:
19
         for (int i = 0; i < 2; i++) {
20
             n = malloc(sizeof(*n));
21
             n->val = 21 - i:
22
             n->next = head:
23
             head = n;
24
25
         print list(head):
26
         n = head; // Explicit cleanup
27
         while (n != NULL) {
28
             node *p = n->next;
29
             free(n);
30
             n = p;
31
32
         return 0;
33
```



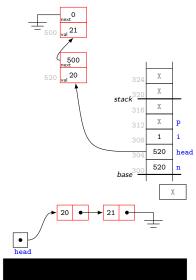
```
typedef struct node {
 4
         int val: // tuped pauload of fixed size inside struct
 5
         struct node *next:
 6
    } node;
     void print_list(const node *p)
 8
 9
         printf("( ");
10
         while (p != NULL) {
11
             printf("%d ", p->val); // we know the type of val
12
             p = p->next:
13
         printf(")\n"):
14
15
16
     int main(void)
17
18
         node *n. *head = NULL:
19
         for (int i = 0; i < 2; i++) {
20
             n = malloc(sizeof(*n));
21
             n->val = 21 - i:
22
             n->next = head:
23
             head = n;
24
25
         print list(head):
26
         n = head; // Explicit cleanup
27
         while (n != NULL) {
28
             node *p = n->next;
29
             free(n);
30
             n = p;
31
32
         return 0;
33
```



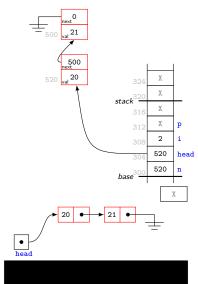
```
typedef struct node {
 4
         int val: // tuped pauload of fixed size inside struct
 5
         struct node *next:
 6
    } node;
     void print_list(const node *p)
 8
 9
         printf("( ");
10
         while (p != NULL) {
11
             printf("%d ", p->val); // we know the type of val
12
             p = p->next:
13
         printf(")\n"):
14
15
16
     int main(void)
17
18
         node *n. *head = NULL:
19
         for (int i = 0; i < 2; i++) {
20
             n = malloc(sizeof(*n));
21
             n->val = 21 - i:
22
             n->next = head:
23
             head = n;
24
25
         print list(head):
26
         n = head; // Explicit cleanup
27
         while (n != NULL) {
28
             node *p = n->next;
29
             free(n);
30
             n = p;
31
32
         return 0;
33
```



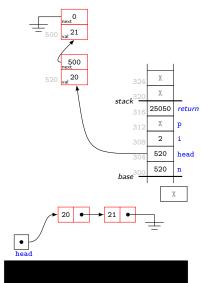
```
typedef struct node {
 4
         int val: // tuped pauload of fixed size inside struct
 5
         struct node *next:
 6
    } node;
     void print_list(const node *p)
 8
 9
         printf("( ");
10
         while (p != NULL) {
11
             printf("%d ", p->val); // we know the type of val
12
             p = p->next:
13
14
         printf(")\n"):
15
16
     int main(void)
17
18
         node *n. *head = NULL:
19
         for (int i = 0; i < 2; i++) {
20
             n = malloc(sizeof(*n));
21
             n->val = 21 - i:
22
             n->next = head:
23
             head = n;
24
25
         print list(head):
26
         n = head; // Explicit cleanup
27
         while (n != NULL) {
28
             node *p = n->next;
29
             free(n);
30
             n = p;
31
32
         return 0;
33
```



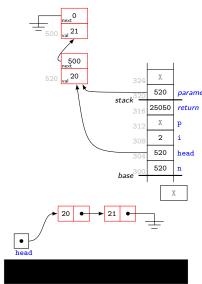
```
typedef struct node {
 4
         int val: // tuped pauload of fixed size inside struct
 5
         struct node *next:
 6
    } node;
     void print_list(const node *p)
 8
 9
         printf("( ");
10
         while (p != NULL) {
11
             printf("%d ", p->val); // we know the type of val
12
             p = p->next:
13
         printf(")\n"):
14
15
16
     int main(void)
17
18
         node *n. *head = NULL:
19
         for (int i = 0; i < 2; i++) {
20
             n = malloc(sizeof(*n));
21
             n->val = 21 - i:
22
             n->next = head:
23
             head = n;
24
25
         print list(head):
26
         n = head; // Explicit cleanup
27
         while (n != NULL) {
28
             node *p = n->next:
29
             free(n);
30
             n = p;
31
32
         return 0;
33
```



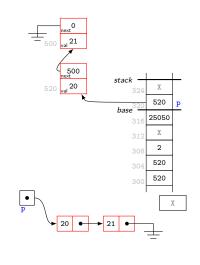
```
typedef struct node {
 4
         int val: // tuped pauload of fixed size inside struct
 5
         struct node *next:
 6
    } node;
     void print_list(const node *p)
 8
 9
         printf("( ");
10
         while (p != NULL) {
11
             printf("%d ", p->val); // we know the type of val
12
             p = p->next:
13
14
         printf(")\n"):
15
16
     int main(void)
17
18
         node *n. *head = NULL:
19
         for (int i = 0; i < 2; i++) {
20
             n = malloc(sizeof(*n));
21
             n->val = 21 - i:
22
             n->next = head:
23
             head = n;
24
25
         print list(head):
26
         n = head; // Explicit cleanup
27
         while (n != NULL) {
28
             node *p = n->next:
29
             free(n);
30
             n = p;
31
32
         return 0;
33
```



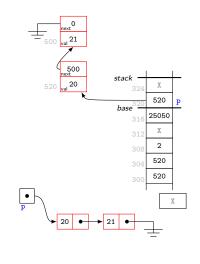
```
typedef struct node {
         int val: // tuped pauload of fixed size inside struct
 5
         struct node *next:
 6
    } node;
     void print_list(const node *p)
 8
 9
         printf("( ");
10
         while (p != NULL) {
11
             printf("%d ", p->val); // we know the type of val
12
             p = p->next:
13
14
         printf(")\n"):
15
16
     int main(void)
17
18
         node *n. *head = NULL:
19
         for (int i = 0; i < 2; i++) {
20
             n = malloc(sizeof(*n));
21
             n->val = 21 - i:
22
             n->next = head:
23
             head = n;
24
25
         print list(head):
26
         n = head; // Explicit cleanup
27
         while (n != NULL) {
28
             node *p = n->next:
29
             free(n);
30
             n = p;
31
32
         return 0;
33
```



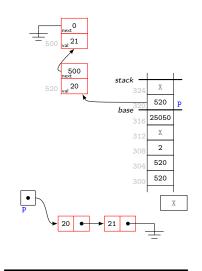
```
typedef struct node {
 4
         int val: // tuped pauload of fixed size inside struct
 5
         struct node *next:
 6
     } node;
     void print_list(const node *p)
 8
 9
         printf("( ");
10
         while (p != NULL) {
11
             printf("%d ", p->val); // we know the type of val
12
             p = p->next:
13
14
         printf(")\n"):
15
16
     int main(void)
17
18
         node *n. *head = NULL:
19
         for (int i = 0; i < 2; i++) {
20
             n = malloc(sizeof(*n));
21
             n->val = 21 - i:
22
             n->next = head:
23
             head = n;
24
25
         print list(head):
26
         n = head; // Explicit cleanup
27
         while (n != NULL) {
28
             node *p = n->next:
29
             free(n);
30
             n = p;
31
32
         return 0;
33
```



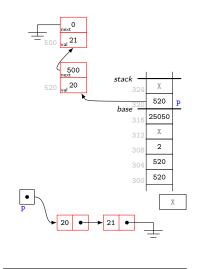
```
typedef struct node {
 4
         int val: // tuped pauload of fixed size inside struct
 5
         struct node *next:
 6
     } node;
     void print_list(const node *p)
 8
 9
         printf("( ");
10
         while (p != NULL) {
11
             printf("%d ", p->val); // we know the type of val
12
             p = p->next:
13
14
         printf(")\n"):
15
16
     int main(void)
17
18
         node *n. *head = NULL:
19
         for (int i = 0; i < 2; i++) {
20
             n = malloc(sizeof(*n));
21
             n->val = 21 - i:
22
             n->next = head:
23
             head = n;
24
25
         print list(head):
26
         n = head; // Explicit cleanup
27
         while (n != NULL) {
28
             node *p = n->next:
29
             free(n);
30
             n = p;
31
32
         return 0;
33
```



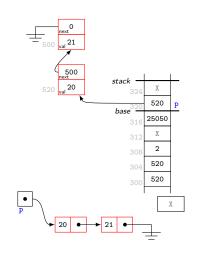
```
typedef struct node {
 4
         int val: // tuped pauload of fixed size inside struct
 5
         struct node *next:
 6
     } node;
     void print_list(const node *p)
 8
 9
         printf("( ");
10
         while (p != NULL) {
11
             printf("%d ", p->val); // we know the type of val
12
             p = p->next:
13
14
         printf(")\n"):
15
16
     int main(void)
17
18
         node *n. *head = NULL:
19
         for (int i = 0; i < 2; i++) {
20
             n = malloc(sizeof(*n));
21
             n->val = 21 - i:
22
             n->next = head:
23
             head = n;
24
25
         print list(head):
26
         n = head; // Explicit cleanup
27
         while (n != NULL) {
28
             node *p = n->next:
29
             free(n);
30
             n = p;
31
32
         return 0;
33
```



```
typedef struct node {
 4
         int val: // tuped pauload of fixed size inside struct
 5
         struct node *next:
 6
     } node;
     void print_list(const node *p)
 8
 9
         printf("( ");
10
         while (p != NULL) {
11
             printf("%d ", p->val); // we know the type of val
12
             p = p->next:
13
14
         printf(")\n"):
15
16
     int main(void)
17
18
         node *n. *head = NULL:
19
         for (int i = 0; i < 2; i++) {
20
             n = malloc(sizeof(*n));
21
             n->val = 21 - i:
22
             n->next = head:
23
             head = n;
24
25
         print list(head):
26
         n = head; // Explicit cleanup
27
         while (n != NULL) {
28
             node *p = n->next:
29
             free(n);
30
             n = p;
31
32
         return 0;
33
```

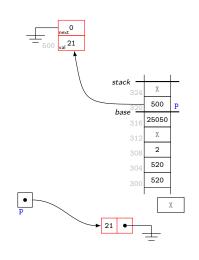


```
typedef struct node {
 4
         int val: // tuped pauload of fixed size inside struct
 5
         struct node *next:
 6
     } node;
     void print_list(const node *p)
 8
 9
         printf("( ");
10
         while (p != NULL) {
11
             printf("%d ", p->val); // we know the type of val
12
             p = p->next:
13
14
         printf(")\n"):
15
16
     int main(void)
17
18
         node *n. *head = NULL:
19
         for (int i = 0; i < 2; i++) {
20
             n = malloc(sizeof(*n));
21
             n->val = 21 - i:
22
             n->next = head:
23
             head = n;
24
25
         print list(head):
26
         n = head; // Explicit cleanup
27
         while (n != NULL) {
28
             node *p = n->next:
29
             free(n);
30
             n = p;
31
32
         return 0;
33
```



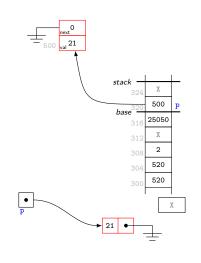
(20

```
typedef struct node {
 4
         int val: // tuped pauload of fixed size inside struct
 5
         struct node *next:
 6
     } node;
     void print_list(const node *p)
 8
 9
         printf("( ");
10
         while (p != NULL) {
11
             printf("%d ", p->val); // we know the type of val
12
             p = p->next:
13
14
         printf(")\n"):
15
16
     int main(void)
17
18
         node *n. *head = NULL:
19
         for (int i = 0; i < 2; i++) {
20
             n = malloc(sizeof(*n));
21
             n->val = 21 - i:
22
             n->next = head:
23
             head = n;
24
25
         print list(head):
26
         n = head; // Explicit cleanup
27
         while (n != NULL) {
28
             node *p = n->next:
29
             free(n);
30
             n = p;
31
32
         return 0;
33
```

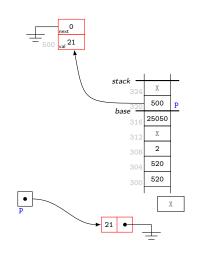


(20

```
typedef struct node {
 4
         int val: // tuped pauload of fixed size inside struct
 5
         struct node *next:
 6
     } node;
     void print_list(const node *p)
 8
 9
         printf("( ");
10
         while (p != NULL) {
11
             printf("%d ", p->val); // we know the type of val
12
             p = p->next:
13
14
         printf(")\n"):
15
16
     int main(void)
17
18
         node *n. *head = NULL:
19
         for (int i = 0; i < 2; i++) {
20
             n = malloc(sizeof(*n));
21
             n->val = 21 - i:
22
             n->next = head:
23
             head = n;
24
25
         print list(head):
26
         n = head; // Explicit cleanup
27
         while (n != NULL) {
28
             node *p = n->next:
29
             free(n);
30
             n = p;
31
32
         return 0;
33
```

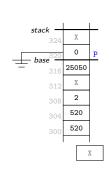


```
typedef struct node {
 4
         int val: // tuped pauload of fixed size inside struct
 5
         struct node *next:
 6
     } node;
     void print_list(const node *p)
 8
 9
         printf("( ");
10
         while (p != NULL) {
11
             printf("%d ", p->val); // we know the type of val
12
             p = p->next:
13
14
         printf(")\n"):
15
16
     int main(void)
17
18
         node *n. *head = NULL:
19
         for (int i = 0; i < 2; i++) {
20
             n = malloc(sizeof(*n));
21
             n->val = 21 - i:
22
             n->next = head:
23
             head = n;
24
25
         print list(head):
26
         n = head; // Explicit cleanup
27
         while (n != NULL) {
28
             node *p = n->next:
29
             free(n);
30
             n = p;
31
32
         return 0;
33
```



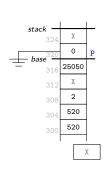
(20 21

```
typedef struct node {
 4
         int val: // tuped pauload of fixed size inside struct
 5
         struct node *next:
 6
     } node;
     void print_list(const node *p)
 8
 9
         printf("( ");
10
         while (p != NULL) {
11
             printf("%d ", p->val); // we know the type of val
12
             p = p->next:
13
14
         printf(")\n"):
15
16
     int main(void)
17
18
         node *n. *head = NULL:
19
         for (int i = 0; i < 2; i++) {
20
             n = malloc(sizeof(*n));
21
             n->val = 21 - i:
22
             n->next = head:
23
             head = n;
24
25
         print list(head):
26
         n = head; // Explicit cleanup
27
         while (n != NULL) {
28
             node *p = n->next:
29
             free(n);
30
             n = p;
31
32
         return 0;
33
```



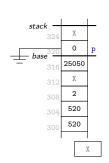
(20 21

```
typedef struct node {
 4
         int val: // tuped pauload of fixed size inside struct
 5
         struct node *next:
 6
    } node;
     void print_list(const node *p)
 8
 9
         printf("( ");
10
         while (p != NULL) {
11
             printf("%d ", p->val); // we know the type of val
12
             p = p->next:
13
         printf(")\n");
14
15
16
     int main(void)
17
18
         node *n. *head = NULL:
19
         for (int i = 0; i < 2; i++) {
20
             n = malloc(sizeof(*n));
21
             n->val = 21 - i:
22
             n->next = head:
23
             head = n;
24
25
         print list(head):
26
         n = head; // Explicit cleanup
27
         while (n != NULL) {
28
             node *p = n->next;
29
             free(n);
30
             n = p;
31
32
         return 0;
33
```



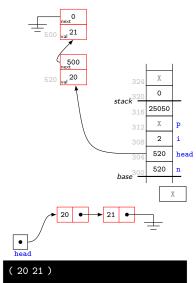
(20 21

```
typedef struct node {
 4
         int val: // tuped pauload of fixed size inside struct
 5
         struct node *next:
 6
    } node;
     void print_list(const node *p)
 8
 9
         printf("( ");
10
         while (p != NULL) {
11
             printf("%d ", p->val); // we know the type of val
12
             p = p->next:
13
14
         printf(")\n");
15
16
     int main(void)
17
18
         node *n. *head = NULL:
19
         for (int i = 0; i < 2; i++) {
20
             n = malloc(sizeof(*n));
21
             n->val = 21 - i:
22
             n->next = head:
23
             head = n;
24
25
         print list(head):
26
         n = head; // Explicit cleanup
27
         while (n != NULL) {
28
             node *p = n->next:
29
             free(n);
30
             n = p;
31
32
         return 0;
33
```

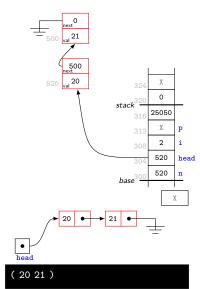


(20 21)

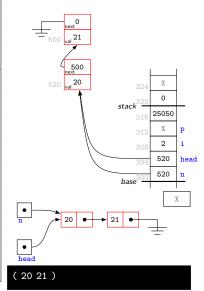
```
typedef struct node {
 4
         int val: // tuped pauload of fixed size inside struct
 5
         struct node *next:
 6
    } node;
     void print_list(const node *p)
 8
 9
         printf("( ");
10
         while (p != NULL) {
11
             printf("%d ", p->val); // we know the type of val
12
             p = p->next:
13
14
         printf(")\n"):
15
16
     int main(void)
17
18
         node *n. *head = NULL:
19
         for (int i = 0; i < 2; i++) {
20
             n = malloc(sizeof(*n));
21
             n->val = 21 - i:
22
             n->next = head:
23
             head = n;
24
25
         print list(head):
26
         n = head; // Explicit cleanup
27
         while (n != NULL) {
28
             node *p = n->next:
29
             free(n);
30
             n = p;
31
32
         return 0;
33
```



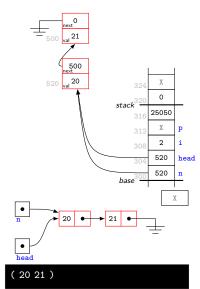
```
typedef struct node {
 4
         int val: // tuped pauload of fixed size inside struct
 5
         struct node *next:
 6
    } node;
     void print_list(const node *p)
 8
 9
         printf("( ");
10
         while (p != NULL) {
11
             printf("%d ", p->val); // we know the type of val
12
             p = p->next:
13
14
         printf(")\n"):
15
16
     int main(void)
17
18
         node *n. *head = NULL:
19
         for (int i = 0; i < 2; i++) {
20
             n = malloc(sizeof(*n));
21
            n->val = 21 - i:
22
             n->next = head:
23
             head = n;
24
25
         print list(head):
26
         n = head; // Explicit cleanup
27
         while (n != NULL) {
28
             node *p = n->next;
29
             free(n);
30
             n = p;
31
32
         return 0;
33
```



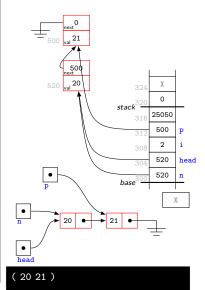
```
typedef struct node {
 4
         int val: // tuped pauload of fixed size inside struct
 5
         struct node *next:
 6
    } node;
     void print_list(const node *p)
 8
 9
         printf("( ");
10
         while (p != NULL) {
11
             printf("%d ", p->val); // we know the type of val
12
             p = p->next:
13
14
         printf(")\n"):
15
16
     int main(void)
17
18
         node *n. *head = NULL:
19
         for (int i = 0; i < 2; i++) {
20
             n = malloc(sizeof(*n));
21
             n->val = 21 - i:
22
             n->next = head:
23
             head = n;
24
25
         print list(head):
26
         n = head; // Explicit cleanup
27
         while (n != NULL) {
28
             node *p = n->next:
29
             free(n);
30
             n = p;
31
32
         return 0;
33
```



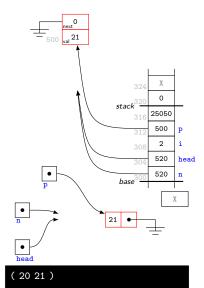
```
typedef struct node {
 4
         int val: // tuped pauload of fixed size inside struct
 5
         struct node *next:
 6
    } node;
     void print_list(const node *p)
 8
 9
         printf("( ");
10
         while (p != NULL) {
11
             printf("%d ", p->val); // we know the type of val
12
             p = p->next:
13
14
         printf(")\n"):
15
16
     int main(void)
17
18
         node *n. *head = NULL:
19
         for (int i = 0; i < 2; i++) {
20
             n = malloc(sizeof(*n));
21
             n->val = 21 - i:
22
             n->next = head:
23
             head = n;
24
25
         print list(head):
26
         n = head; // Explicit cleanup
27
         while (n != NULL) {
28
             node *p = n->next;
29
             free(n);
30
             n = p;
31
32
         return 0;
33
```



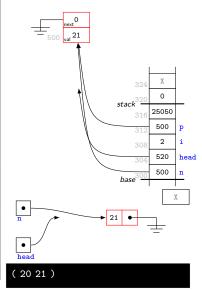
```
typedef struct node {
 4
         int val: // tuped pauload of fixed size inside struct
 5
         struct node *next:
 6
    } node;
     void print_list(const node *p)
 8
 9
         printf("( ");
10
         while (p != NULL) {
11
             printf("%d ", p->val); // we know the type of val
12
             p = p->next:
13
         printf(")\n"):
14
15
16
     int main(void)
17
18
         node *n. *head = NULL:
19
         for (int i = 0; i < 2; i++) {
20
             n = malloc(sizeof(*n));
21
             n->val = 21 - i:
22
             n->next = head:
23
             head = n;
24
25
         print list(head):
26
         n = head; // Explicit cleanup
27
         while (n != NULL) {
28
             node *p = n->next;
29
             free(n);
30
             n = p;
31
32
         return 0;
33
```



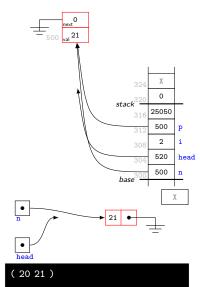
```
typedef struct node {
 4
         int val: // tuped pauload of fixed size inside struct
 5
         struct node *next:
 6
    } node;
     void print_list(const node *p)
 8
 9
         printf("( ");
10
         while (p != NULL) {
11
             printf("%d ", p->val); // we know the type of val
12
             p = p->next:
13
         printf(")\n"):
14
15
16
     int main(void)
17
18
         node *n. *head = NULL:
19
         for (int i = 0; i < 2; i++) {
20
             n = malloc(sizeof(*n));
21
             n->val = 21 - i:
22
             n->next = head:
23
             head = n;
24
25
         print list(head):
26
         n = head; // Explicit cleanup
27
         while (n != NULL) {
28
             node *p = n->next:
29
             free(n);
30
             n = p;
31
32
         return 0;
33
```



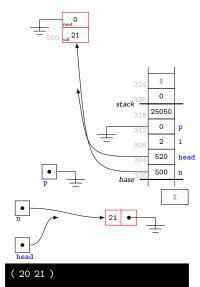
```
typedef struct node {
 4
         int val: // tuped pauload of fixed size inside struct
 5
         struct node *next:
 6
    } node;
     void print_list(const node *p)
 8
 9
         printf("( ");
10
         while (p != NULL) {
11
             printf("%d ", p->val); // we know the type of val
12
             p = p->next:
13
14
         printf(")\n"):
15
16
     int main(void)
17
18
         node *n. *head = NULL:
19
         for (int i = 0; i < 2; i++) {
20
             n = malloc(sizeof(*n));
21
             n->val = 21 - i:
22
             n->next = head:
23
             head = n;
24
25
         print list(head):
26
         n = head; // Explicit cleanup
27
         while (n != NULL) {
28
             node *p = n->next:
29
             free(n);
30
             n = p;
31
32
         return 0;
33
```



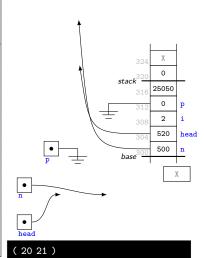
```
typedef struct node {
 4
         int val: // tuped pauload of fixed size inside struct
 5
         struct node *next:
 6
    } node;
     void print_list(const node *p)
 8
 9
         printf("( ");
10
         while (p != NULL) {
11
             printf("%d ", p->val); // we know the type of val
12
             p = p->next:
13
14
         printf(")\n"):
15
16
     int main(void)
17
18
         node *n. *head = NULL:
19
         for (int i = 0; i < 2; i++) {
20
             n = malloc(sizeof(*n));
21
             n->val = 21 - i:
22
             n->next = head:
23
             head = n;
24
25
         print list(head):
26
         n = head; // Explicit cleanup
27
         while (n != NULL) {
28
             node *p = n->next;
29
             free(n);
30
             n = p;
31
32
         return 0;
33
```



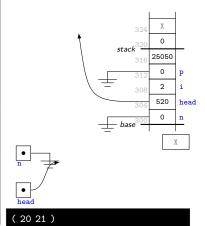
```
typedef struct node {
 4
         int val: // tuped pauload of fixed size inside struct
 5
         struct node *next:
 6
    } node;
     void print_list(const node *p)
 8
 9
         printf("( ");
10
         while (p != NULL) {
11
             printf("%d ", p->val); // we know the type of val
12
             p = p->next:
13
         printf(")\n"):
14
15
16
     int main(void)
17
18
         node *n. *head = NULL:
19
         for (int i = 0; i < 2; i++) {
20
             n = malloc(sizeof(*n));
21
             n->val = 21 - i:
22
             n->next = head:
23
             head = n;
24
25
         print list(head):
26
         n = head; // Explicit cleanup
27
         while (n != NULL) {
28
             node *p = n->next;
29
             free(n);
30
             n = p;
31
32
         return 0;
33
```



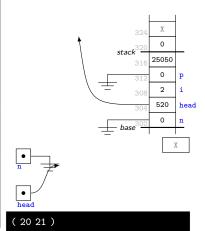
```
typedef struct node {
 4
         int val: // tuped pauload of fixed size inside struct
 5
         struct node *next:
 6
    } node;
     void print_list(const node *p)
 8
 9
         printf("( ");
10
         while (p != NULL) {
11
             printf("%d ", p->val); // we know the type of val
12
             p = p->next:
13
14
         printf(")\n"):
15
16
     int main(void)
17
18
         node *n. *head = NULL:
19
         for (int i = 0; i < 2; i++) {
20
             n = malloc(sizeof(*n));
21
             n->val = 21 - i:
22
             n->next = head:
23
             head = n;
24
25
         print list(head):
26
         n = head; // Explicit cleanup
27
         while (n != NULL) {
28
             node *p = n->next:
29
             free(n);
30
             n = p;
31
32
         return 0;
33
```



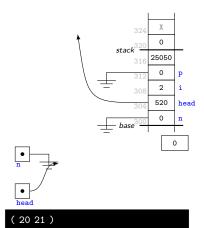
```
typedef struct node {
 4
         int val: // tuped pauload of fixed size inside struct
 5
         struct node *next:
 6
    } node;
     void print_list(const node *p)
 8
 9
         printf("( ");
10
         while (p != NULL) {
11
             printf("%d ", p->val); // we know the type of val
12
             p = p->next:
13
14
         printf(")\n"):
15
16
     int main(void)
17
18
         node *n. *head = NULL:
19
         for (int i = 0; i < 2; i++) {
20
             n = malloc(sizeof(*n));
21
             n->val = 21 - i:
22
             n->next = head:
23
             head = n;
24
25
         print list(head):
26
         n = head; // Explicit cleanup
27
         while (n != NULL) {
28
             node *p = n->next:
29
             free(n);
30
             n = p;
31
32
         return 0;
33
```



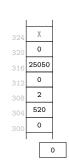
```
typedef struct node {
 4
         int val: // tuped pauload of fixed size inside struct
 5
         struct node *next:
 6
    } node;
     void print_list(const node *p)
 8
 9
         printf("( ");
10
         while (p != NULL) {
11
             printf("%d ", p->val); // we know the type of val
12
             p = p->next:
13
14
         printf(")\n"):
15
16
     int main(void)
17
18
         node *n. *head = NULL:
19
         for (int i = 0; i < 2; i++) {
20
             n = malloc(sizeof(*n));
21
             n->val = 21 - i:
22
             n->next = head:
23
             head = n;
24
25
         print list(head):
26
         n = head; // Explicit cleanup
27
         while (n != NULL) {
28
             node *p = n->next;
29
             free(n);
30
             n = p;
31
32
         return 0;
33
```



```
typedef struct node {
 4
         int val: // tuped pauload of fixed size inside struct
 5
         struct node *next:
 6
    } node;
     void print_list(const node *p)
 8
 9
         printf("( ");
10
         while (p != NULL) {
11
             printf("%d ", p->val); // we know the type of val
12
             p = p->next:
13
14
         printf(")\n"):
15
16
     int main(void)
17
18
         node *n. *head = NULL:
19
         for (int i = 0; i < 2; i++) {
20
             n = malloc(sizeof(*n));
21
             n->val = 21 - i:
22
             n->next = head:
23
             head = n;
24
25
         print list(head):
26
         n = head; // Explicit cleanup
27
         while (n != NULL) {
28
             node *p = n->next;
29
             free(n);
30
             n = p;
31
32
         return 0;
33
```



```
typedef struct node {
 4
         int val: // tuped pauload of fixed size inside struct
 5
         struct node *next:
 6
    } node;
     void print_list(const node *p)
 8
 9
         printf("( ");
10
         while (p != NULL) {
11
             printf("%d ", p->val); // we know the type of val
12
             p = p->next:
13
14
         printf(")\n"):
15
16
     int main(void)
17
18
         node *n. *head = NULL:
19
         for (int i = 0; i < 2; i++) {
20
             n = malloc(sizeof(*n));
21
            n->val = 21 - i:
22
             n->next = head:
23
             head = n;
24
25
         print list(head):
26
         n = head; // Explicit cleanup
27
         while (n != NULL) {
28
             node *p = n->next;
29
             free(n);
30
             n = p;
31
32
         return 0;
33
```



(20 21)

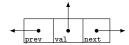
Generiska datatyper i C

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:

```
32 typedef struct cell {
33 struct cell *next;
34 struct cell *prev;
35 void *val;
36 } 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

Hur kodbasen hanterar abstraktion (2)

Huvudprogrammet inkluderar list.h

```
code/list-with-freehandler.c _______3 #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
       int main(void)
19
20
           // Create empty list, hand over responsibility to
21
22
           // deallocate payload using int kill
23
           list *1 = list_empty(int_kill);
24
           list_pos p = list_first(1);
25
           for (int i = 0; i < 2; i++) {
26
               int *v = int_create(20 + i);
27
               p = list_insert(1, v, p);
28
               p = list next(1, p):
29
30
           list print(1, print int):
31
32
33
34
35
36
           // Clean up the list, including payload
37
38
           list kill(1):
39
           return 0;
40
```

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
27
28
        * The list elements are implemented as two-cells with forward and
29
        * backward links and a void * for the value. The list uses two border
30
        * cells at the start and end of the list.
31
        */
32
       typedef struct cell {
33
           struct cell *next;
34
           struct cell *prev;
35
           void *val:
36
       } cell;
37
38
       struct list {
39
           cell *head:
40
           cell *tail:
41
           kill function kill func:
42
       };
```

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

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 _
163
        list_pos list_insert(list *1, void *v, const list_pos p)
164
165
            // Allocate memory for a new cell.
166
            list_pos e = malloc(sizeof(cell));
167
168
            // Store the value.
            e->val = v;
169
170
            // Add links to/from the new cell.
171
            e->next = p:
172
            e->prev = p->prev;
173
            e->prev->next = e;
174
            p->prev = e:
175
176
            // Return the position of the new cell.
177
            return e;
178
```

► 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

int *int_create(int v) {

int *p = malloc(sizeof(*p)); // allocate payload

*p = v; // assign value

return p;
}
```

```
code/list-with-freehandler.c
19
       int main(void)
20
21
           // Create empty list, hand over responsibility to
22
           // deallocate payload using int kill
23
           list *l = list_empty(int_kill);
24
           list pos p = list first(1):
25
           for (int i = 0: i < 2: i++) {
26
               int *v = int_create(20 + i);
27
               p = list insert(1, v, p):
               p = list_next(1, p);
28
29
30
           list_print(1, print_int);
```

Hur kodbasen hanterar abstraktion (6)

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

```
#include <stdio h>
       #include <stdlih h>
       #include <liist.h>
       void print_int(const void *p)
 6
           const int *q = p;
           printf("%d ", *q);
 8
       int *int create(int v)
10
11
           int *p = malloc(sizeof(*p)); // allocate payload
12
           *p = v; // assign value
13
           return p;
14
15
       void int_kill(void *v)
16
17
           free(v); // deallocate payload
18
```

Hjälpfunktionerna är:

Hur kodbasen hanterar abstraktion (7)

► Huvudprogrammet ser ut så här:

```
code/list-with-killfunction.c
19
       int main(void)
20
       ſ
21
           // Create empty list, hand over responsibility to
22
           // deallocate payload using int kill
23
           list *1 = list empty(int kill):
24
           list_pos p = list_first(1);
25
           for (int i = 0; i < 2; i++) {
26
               int *v = int_create(20 + i);
27
               p = list_insert(1, v, p);
28
               p = list_next(1, p);
29
30
           list_print(1, print_int);
31
32
33
34
35
36
37
           // Clean up the list, including payload
38
           list_kill(1);
39
           return 0;
40
```

- ► 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
       int main(void)
19
20
           // Create empty list, hand over responsibility to
21
22
           // deallocate payload using int kill
23
           list *1 = list empty(int kill):
24
           list pos p = list first(1):
25
           for (int i = 0; i < 2; i++) {
26
               int *v = int_create(20 + i);
27
               p = list_insert(1, v, p);
28
               p = list_next(1, p);
29
30
           list_print(1, print_int);
31
```

- ► 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
19
       int main(void)
20
           // Create empty list, hand over responsibility to
21
22
           // deallocate payload using int kill
23
           list *1 = list empty(int kill):
24
           list pos p = list first(1):
25
           for (int i = 0; i < 2; i++) {
26
               int *v = int_create(20 + i);
27
               p = list_insert(1, v, p);
28
               p = list_next(1, p);
29
30
           list_print(1, print_int);
31
32
33
34
35
36
37
           // Clean up the list, including payload
38
           list_kill(1);
39
           return 0;
40
```

```
int *int_create(int v)
10
11
         int *p = malloc(sizeof(*p)); // allocate payload
12
        *p = v; // assign value
13
         return p;
14
15
     void int_kill(void *v)
16
17
         free(v); // deallocate payload
18
19
     int main(void)
20
     {
21
         // Create empty list, hand over responsibility to
22
         // deallocate payload using int kill
23
         list *l = list_empty(int_kill);
24
         list_pos p = list_first(1);
25
         for (int i = 0; i < 2; i++) {
26
             int *v = int_create(20 + i);
27
             p = list_insert(1, v, p);
28
             p = list next(1, p):
29
30
         list_print(1, print_int);
31
32
33
34
35
36
37
         // Clean up the list, including payload
38
         list_kill(1);
39
         return 0;
40
```

```
X
X
X
X
X
X
X
```

```
int *int_create(int v)
10
11
         int *p = malloc(sizeof(*p)); // allocate payload
12
         *p = v; // assign value
13
         return p;
14
15
     void int_kill(void *v)
16
17
         free(v); // deallocate payload
18
19
     int main(void)
20
21
         // Create empty list, hand over responsibility to
22
         // deallocate payload using int kill
23
         list *l = list_empty(int_kill);
24
         list_pos p = list_first(1);
25
         for (int i = 0; i < 2; i++) {
26
             int *v = int_create(20 + i);
27
             p = list_insert(1, v, p);
28
             p = list next(1, p):
29
30
         list_print(1, print_int);
31
32
33
34
35
36
37
         // Clean up the list, including payload
38
         list_kill(1);
39
         return 0;
40
```

```
X
X
X
X
X
X
X
X
```

```
int *int_create(int v)
10
11
         int *p = malloc(sizeof(*p)); // allocate payload
12
        *p = v; // assign value
13
         return p;
14
15
     void int_kill(void *v)
16
17
         free(v); // deallocate payload
18
19
     int main(void)
20
     {
21
         // Create empty list, hand over responsibility to
22
         // deallocate payload using int kill
23
         list *1 = list_empty(int_kill);
24
         list_pos p = list_first(1);
25
         for (int i = 0; i < 2; i++) {
26
             int *v = int_create(20 + i);
27
             p = list_insert(1, v, p);
28
             p = list next(1, p):
29
30
         list_print(1, print_int);
31
32
33
34
35
36
37
         // Clean up the list, including payload
38
         list_kill(1);
39
         return 0;
40
```

```
X
X
X
X
X
X
X
X
```

```
int *int_create(int v)
10
         int *p = malloc(sizeof(*p)); // allocate payload
11
12
        *p = v; // assign value
13
         return p;
14
15
     void int_kill(void *v)
16
17
         free(v); // deallocate payload
18
19
     int main(void)
20
     {
21
         // Create empty list, hand over responsibility to
22
         // deallocate payload using int kill
23
         list *1 = list_empty(int_kill);
24
         list_pos p = list_first(1);
25
         for (int i = 0; i < 2; i++) {
26
             int *v = int_create(20 + i);
27
             p = list_insert(1, v, p);
28
             p = list next(1, p):
29
30
         list_print(1, print_int);
31
32
33
34
35
36
37
         // Clean up the list, including payload
38
         list_kill(1);
39
         return 0;
40
```

```
X
  X
  X
  X
  X
23050
  X
  X
  X
```

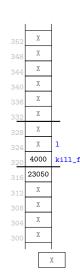
```
int *int_create(int v)
10
         int *p = malloc(sizeof(*p)); // allocate payload
11
12
         *p = v; // assign value
13
         return p;
14
15
     void int_kill(void *v)
16
17
         free(v); // deallocate payload
18
19
     int main(void)
20
     {
21
         // Create empty list, hand over responsibility to
22
         // deallocate payload using int kill
23
         list *1 = list_empty(int_kill);
24
         list_pos p = list_first(1);
25
         for (int i = 0; i < 2; i++) {
26
             int *v = int_create(20 + i);
27
             p = list_insert(1, v, p);
28
             p = list next(1, p):
29
30
         list_print(1, print_int);
31
32
33
34
35
36
37
         // Clean up the list, including payload
38
         list_kill(1);
39
         return 0;
40
```

```
X
  X
  X
  X
4000
        р1
23050
  X
  X
  X
```

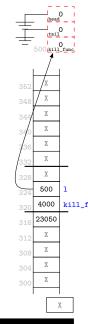
```
48
49
      * list emptu() - Create an emptu list.
50
      * Okill_func: A pointer to a function (or NULL) to be ca
51
                    de-allocate memory on remove/kill.
52
53
      * Returns: A pointer to the new list.
54
      */
55
     list *list empty(kill function kill func)
56
57
         // Allocate memory for the list head.
58
         list *l = calloc(1, sizeof(list));
59
60
         // Allocate memory for the border cells.
61
         1->head = calloc(1, sizeof(cell));
62
         1->tail = calloc(1, sizeof(cell));
63
64
         // Set consistent links between border elements.
65
        1->head->next = 1->tail;
66
         1->tail->prev = 1->head:
67
68
         // Store the kill function.
69
         1->kill func = kill func:
70
71
         return 1;
72
```

```
X
  X
  X
  X
4000
       kill f
23050
  X
  X
  X
```

```
48
49
      * list emptu() - Create an emptu list.
50
      * Okill_func: A pointer to a function (or NULL) to be ca
51
                    de-allocate memory on remove/kill.
52
53
      * Returns: A pointer to the new list.
54
      */
55
     list *list empty(kill function kill func)
56
     {
         // Allocate memory for the list head.
57
58
         list *l = calloc(1, sizeof(list));
59
60
         // Allocate memory for the border cells.
61
         1->head = calloc(1, sizeof(cell));
62
         1->tail = calloc(1, sizeof(cell));
63
64
         // Set consistent links between border elements.
65
         1->head->next = 1->tail;
66
         1->tail->prev = 1->head:
67
68
         // Store the kill function.
69
         1->kill_func = kill_func;
70
71
         return 1;
72
```



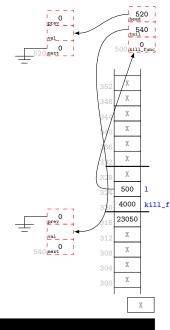
```
48
49
      * list emptu() - Create an emptu list.
50
      * Okill_func: A pointer to a function (or NULL) to be ca
51
                    de-allocate memory on remove/kill.
52
53
      * Returns: A pointer to the new list.
54
      */
55
     list *list empty(kill function kill func)
56
     {
57
        // Allocate memory for the list head.
58
        list *l = calloc(1, sizeof(list));
59
60
         // Allocate memory for the border cells.
61
         1->head = calloc(1, sizeof(cell));
62
         1->tail = calloc(1, sizeof(cell));
63
64
         // Set consistent links between border elements.
65
        1->head->next = 1->tail;
66
         1->tail->prev = 1->head:
67
68
         // Store the kill function.
69
         1->kill_func = kill_func;
70
71
         return 1;
72
```



```
48
49
      * list emptu() - Create an emptu list.
50
      * Okill_func: A pointer to a function (or NULL) to be ca
51
                    de-allocate memory on remove/kill.
52
53
      * Returns: A pointer to the new list.
54
      */
55
     list *list empty(kill function kill func)
56
     {
57
         // Allocate memory for the list head.
58
         list *l = calloc(1, sizeof(list));
59
60
         // Allocate memory for the border cells.
61
         1->head = calloc(1, sizeof(cell));
62
         1->tail = calloc(1, sizeof(cell));
63
64
         // Set consistent links between border elements.
65
         1->head->next = 1->tail;
66
         1->tail->prev = 1->head:
67
68
         // Store the kill function.
69
         1->kill_func = kill_func;
70
71
         return 1;
72
```

```
0
prev
                    Ookill func
                     X
                     X
                     X
                     X
                     χ
                    500
                   4000
                          kill f
                   23050
                     X
                     X
                     X
```

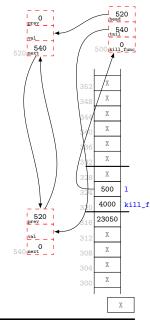
```
48
49
      * list emptu() - Create an emptu list.
50
      * Okill_func: A pointer to a function (or NULL) to be ca
51
                    de-allocate memory on remove/kill.
52
53
      * Returns: A pointer to the new list.
54
      */
55
     list *list empty(kill function kill func)
56
     {
57
         // Allocate memory for the list head.
58
         list *l = calloc(1, sizeof(list));
59
60
         // Allocate memory for the border cells.
61
        1->head = calloc(1, sizeof(cell));
62
         1->tail = calloc(1, sizeof(cell));
63
64
         // Set consistent links between border elements.
65
         1->head->next = 1->tail;
66
         1->tail->prev = 1->head:
67
68
         // Store the kill function.
69
         1->kill func = kill func:
70
71
         return 1;
72
```



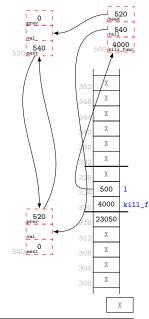
```
48
49
      * list emptu() - Create an emptu list.
50
      * Okill_func: A pointer to a function (or NULL) to be ca
51
                    de-allocate memory on remove/kill.
52
53
      * Returns: A pointer to the new list.
54
      */
55
     list *list empty(kill function kill func)
56
     {
57
         // Allocate memory for the list head.
58
         list *l = calloc(1, sizeof(list));
59
60
         // Allocate memory for the border cells.
61
        1->head = calloc(1, sizeof(cell));
62
         1->tail = calloc(1, sizeof(cell));
63
64
         // Set consistent links between border elements.
65
         1->head->next = 1->tail;
66
         1->tail->prev = 1->head:
67
68
         // Store the kill function.
69
         1->kill func = kill func:
70
71
         return 1;
72
```

```
0
   prev
                        540
                     500 kill_func
     540
520 next
                        X
                        X
                        X
                        X
                        χ
                       500
                      4000
                             kill f
      Ω
                     23050
                        X
                        X
                        X
```

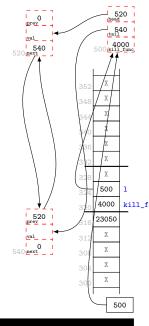
```
48
49
      * list emptu() - Create an emptu list.
50
      * Okill_func: A pointer to a function (or NULL) to be ca
51
                    de-allocate memory on remove/kill.
52
53
      * Returns: A pointer to the new list.
54
      */
55
     list *list empty(kill function kill func)
56
     {
57
        // Allocate memory for the list head.
58
         list *l = calloc(1, sizeof(list));
59
60
         // Allocate memory for the border cells.
61
        1->head = calloc(1, sizeof(cell));
62
         1->tail = calloc(1, sizeof(cell));
63
64
         // Set consistent links between border elements.
65
        1->head->next = 1->tail;
66
         1->tail->prev = 1->head:
67
68
         // Store the kill function.
69
         1->kill_func = kill_func;
70
71
         return 1;
72
```



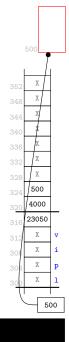
```
48
49
      * list emptu() - Create an emptu list.
50
      * Okill_func: A pointer to a function (or NULL) to be ca
51
                    de-allocate memory on remove/kill.
52
53
      * Returns: A pointer to the new list.
54
      */
55
     list *list empty(kill function kill func)
56
     {
57
         // Allocate memory for the list head.
58
         list *l = calloc(1, sizeof(list));
59
60
         // Allocate memory for the border cells.
61
        1->head = calloc(1, sizeof(cell));
62
         1->tail = calloc(1, sizeof(cell));
63
64
         // Set consistent links between border elements.
65
        1->head->next = 1->tail;
66
         1->tail->prev = 1->head:
67
68
         // Store the kill function.
69
         1->kill func = kill func:
70
71
         return 1;
72
```



```
48
49
      * list emptu() - Create an emptu list.
50
      * Okill_func: A pointer to a function (or NULL) to be ca
51
                    de-allocate memory on remove/kill.
52
53
      * Returns: A pointer to the new list.
54
      */
55
     list *list empty(kill function kill func)
56
     {
57
        // Allocate memory for the list head.
58
         list *l = calloc(1, sizeof(list));
59
60
         // Allocate memory for the border cells.
61
         1->head = calloc(1, sizeof(cell));
62
         1->tail = calloc(1, sizeof(cell));
63
64
         // Set consistent links between border elements.
65
        1->head->next = 1->tail;
66
         1->tail->prev = 1->head:
67
68
         // Store the kill function.
69
         1->kill func = kill func:
70
71
         return 1;
72
```



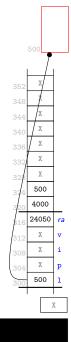
```
int *int_create(int v)
10
11
         int *p = malloc(sizeof(*p)); // allocate payload
12
        *p = v; // assign value
13
         return p;
14
15
     void int_kill(void *v)
16
17
         free(v); // deallocate payload
18
19
     int main(void)
20
     {
21
         // Create empty list, hand over responsibility to
22
         // deallocate payload using int kill
23
         list *1 = list_empty(int_kill);
24
         list_pos p = list_first(1);
25
         for (int i = 0; i < 2; i++) {
26
             int *v = int_create(20 + i);
27
             p = list_insert(1, v, p);
28
             p = list next(1, p):
29
30
         list_print(1, print_int);
31
32
33
34
35
36
37
         // Clean up the list, including payload
38
         list_kill(1);
39
         return 0;
40
```



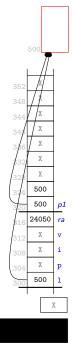
```
int *int_create(int v)
10
11
         int *p = malloc(sizeof(*p)); // allocate payload
12
        *p = v; // assign value
13
         return p;
14
15
     void int_kill(void *v)
16
17
         free(v); // deallocate payload
18
19
     int main(void)
20
     {
21
        // Create empty list, hand over responsibility to
22
        // deallocate payload using int kill
23
         list *l = list_empty(int_kill);
24
         list_pos p = list_first(1);
25
         for (int i = 0; i < 2; i++) {
26
             int *v = int_create(20 + i);
27
             p = list_insert(1, v, p);
28
             p = list next(1, p):
29
30
         list_print(1, print_int);
31
32
33
34
35
36
37
         // Clean up the list, including payload
38
         list_kill(1);
39
         return 0;
40
```

```
X
       X
       χ
     500
32
     4000
    23050
       X
       X
       X
     500
```

```
int *int_create(int v)
10
11
         int *p = malloc(sizeof(*p)); // allocate payload
12
        *p = v; // assign value
13
         return p;
14
15
     void int_kill(void *v)
16
17
         free(v); // deallocate payload
18
19
     int main(void)
20
     {
21
        // Create empty list, hand over responsibility to
22
        // deallocate payload using int kill
23
         list *l = list_empty(int_kill);
24
         list_pos p = list_first(1);
25
         for (int i = 0; i < 2; i++) {
26
             int *v = int_create(20 + i);
27
             p = list_insert(1, v, p);
28
             p = list next(1, p):
29
30
         list_print(1, print_int);
31
32
33
34
35
36
37
         // Clean up the list, including payload
38
         list_kill(1);
39
         return 0;
40
```



```
int *int_create(int v)
10
11
         int *p = malloc(sizeof(*p)); // allocate payload
12
        *p = v; // assign value
13
         return p;
14
15
     void int_kill(void *v)
16
17
         free(v); // deallocate payload
18
19
     int main(void)
20
     {
21
        // Create empty list, hand over responsibility to
22
        // deallocate payload using int kill
23
         list *l = list_empty(int_kill);
24
         list_pos p = list_first(1);
25
         for (int i = 0; i < 2; i++) {
26
             int *v = int_create(20 + i);
27
             p = list_insert(1, v, p);
28
             p = list next(1, p):
29
30
         list_print(1, print_int);
31
32
33
34
35
36
37
         // Clean up the list, including payload
38
         list_kill(1);
39
         return 0;
40
```



86

87

88

89

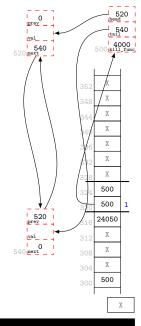
90 91

92 93

94

95 96

```
* 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 *1)
    // First position is position of first element.
    return 1->head->next;
```



86

87

88

89

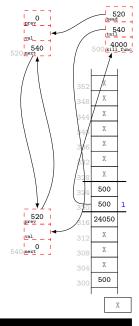
90 91

92 93

94

95 96

```
* list_first() - Return the first position of a list, i.
                  position of the first element in the li
 * Ql: List to inspect.
 * Returns: The first position in the given list.
list_pos list_first(const list *1)
    // First position is position of first element.
    return 1->head->next;
```



86

87

88

89

90 91

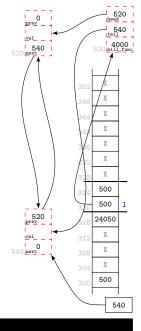
92 93

94

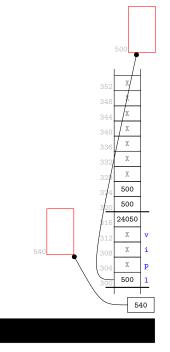
95

96

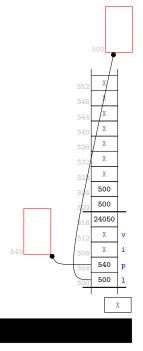
```
* list_first() - Return the first position of a list, i.
                  position of the first element in the li
 * Ql: List to inspect.
 * Returns: The first position in the given list.
list_pos list_first(const list *1)
    // First position is position of first element.
    return 1->head->next;
```



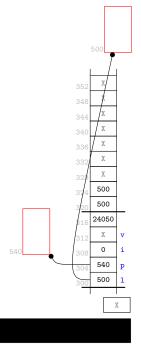
```
int *int_create(int v)
10
11
         int *p = malloc(sizeof(*p)); // allocate payload
12
        *p = v; // assign value
13
         return p;
14
15
     void int_kill(void *v)
16
17
         free(v); // deallocate payload
18
19
     int main(void)
20
     {
21
        // Create empty list, hand over responsibility to
22
        // deallocate payload using int kill
23
         list *l = list_empty(int_kill);
24
         list_pos p = list_first(1);
25
         for (int i = 0; i < 2; i++) {
26
             int *v = int_create(20 + i);
27
             p = list_insert(1, v, p);
28
             p = list next(1, p):
29
30
         list_print(1, print_int);
31
32
33
34
35
36
37
         // Clean up the list, including payload
38
         list_kill(1);
39
         return 0;
40
```



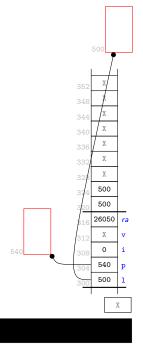
```
int *int_create(int v)
10
11
         int *p = malloc(sizeof(*p)); // allocate payload
12
        *p = v; // assign value
13
         return p;
14
15
     void int_kill(void *v)
16
17
         free(v); // deallocate payload
18
19
     int main(void)
20
     {
21
        // Create empty list, hand over responsibility to
22
         // deallocate payload using int kill
23
         list *l = list_empty(int_kill);
24
         list_pos p = list_first(1);
25
         for (int i = 0; i < 2; i++) {
26
             int *v = int_create(20 + i);
27
             p = list_insert(1, v, p);
28
             p = list next(1, p):
29
30
         list_print(1, print_int);
31
32
33
34
35
36
37
         // Clean up the list, including payload
38
         list_kill(1);
39
         return 0;
40
```



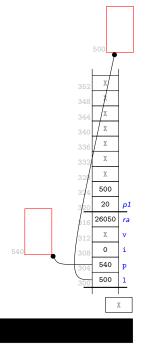
```
int *int_create(int v)
10
11
         int *p = malloc(sizeof(*p)); // allocate payload
12
        *p = v; // assign value
13
         return p;
14
15
     void int_kill(void *v)
16
17
         free(v); // deallocate payload
18
19
     int main(void)
20
     {
21
         // Create empty list, hand over responsibility to
22
         // deallocate payload using int kill
23
         list *l = list_empty(int_kill);
24
         list_pos p = list_first(1);
25
         for (int i = 0; i < 2; i++) {
26
             int *v = int_create(20 + i);
27
             p = list_insert(1, v, p);
28
             p = list next(1, p):
29
30
         list_print(1, print_int);
31
32
33
34
35
36
37
         // Clean up the list, including payload
38
         list_kill(1);
39
         return 0;
40
```



```
int *int_create(int v)
10
11
         int *p = malloc(sizeof(*p)); // allocate payload
12
        *p = v; // assign value
13
         return p;
14
15
     void int_kill(void *v)
16
17
         free(v); // deallocate payload
18
19
     int main(void)
20
     {
21
         // Create empty list, hand over responsibility to
22
         // deallocate payload using int kill
23
         list *l = list_empty(int_kill);
24
         list_pos p = list_first(1);
25
         for (int i = 0; i < 2; i++) {
26
             int *v = int_create(20 + i);
27
             p = list_insert(1, v, p);
28
             p = list next(1, p):
29
30
         list_print(1, print_int);
31
32
33
34
35
36
37
         // Clean up the list, including payload
38
         list_kill(1);
39
         return 0;
40
```



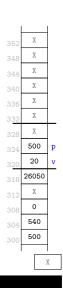
```
int *int_create(int v)
10
11
         int *p = malloc(sizeof(*p)); // allocate payload
12
        *p = v; // assign value
13
         return p;
14
15
     void int_kill(void *v)
16
17
         free(v); // deallocate payload
18
19
     int main(void)
20
     {
21
         // Create empty list, hand over responsibility to
22
         // deallocate payload using int kill
23
         list *l = list_empty(int_kill);
24
         list_pos p = list_first(1);
25
         for (int i = 0; i < 2; i++) {
26
             int *v = int_create(20 + i);
27
             p = list_insert(1, v, p);
28
             p = list next(1, p):
29
30
         list_print(1, print_int);
31
32
33
34
35
36
37
         // Clean up the list, including payload
38
         list_kill(1);
39
         return 0;
40
```



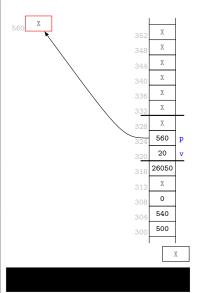
```
int *int_create(int v)
10
         int *p = malloc(sizeof(*p)); // allocate payload
11
12
         *p = v; // assign value
13
         return p;
14
15
     void int_kill(void *v)
16
17
         free(v); // deallocate payload
18
19
     int main(void)
20
     {
21
         // Create empty list, hand over responsibility to
22
         // deallocate payload using int kill
23
         list *l = list_empty(int_kill);
24
         list_pos p = list_first(1);
25
         for (int i = 0; i < 2; i++) {
26
             int *v = int_create(20 + i);
27
             p = list_insert(1, v, p);
28
             p = list next(1, p):
29
30
         list_print(1, print_int);
31
32
33
34
35
36
37
         // Clean up the list, including payload
38
         list_kill(1);
39
         return 0;
40
```

```
X
  X
  X
  X
  χ
 500
  20
26050
  0
 540
 500
```

```
int *int_create(int v)
10
11
         int *p = malloc(sizeof(*p)); // allocate payload
12
         *p = v; // assign value
13
         return p;
14
    }
15
     void int_kill(void *v)
16
17
         free(v); // deallocate payload
18
19
     int main(void)
20
     {
21
         // Create empty list, hand over responsibility to
22
         // deallocate payload using int kill
23
         list *l = list_empty(int_kill);
24
         list_pos p = list_first(1);
25
         for (int i = 0; i < 2; i++) {
26
             int *v = int_create(20 + i);
27
             p = list_insert(1, v, p);
28
             p = list next(1, p):
29
30
         list_print(1, print_int);
31
32
33
34
35
36
37
         // Clean up the list, including payload
38
         list_kill(1);
39
         return 0;
40
```



```
int *int_create(int v)
10
11
         int *p = malloc(sizeof(*p)); // allocate payload
12
         *p = v; // assign value
13
         return p;
14
15
     void int_kill(void *v)
16
17
         free(v); // deallocate payload
18
19
     int main(void)
20
     {
21
         // Create empty list, hand over responsibility to
22
         // deallocate payload using int kill
23
         list *l = list_empty(int_kill);
24
         list_pos p = list_first(1);
25
         for (int i = 0; i < 2; i++) {
26
             int *v = int_create(20 + i);
27
             p = list_insert(1, v, p);
28
             p = list next(1, p):
29
30
         list_print(1, print_int);
31
32
33
34
35
36
37
         // Clean up the list, including payload
38
         list_kill(1);
39
         return 0;
40
```



10 11

12

13

14 15

16 17

18 19

20

21

22

23

24

25

26

27

28

29 30

31

32

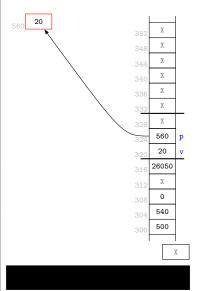
33 34

 $\frac{35}{36}$

38

39

```
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(1);
    for (int i = 0; i < 2; i++) {
        int *v = int_create(20 + i);
        p = list_insert(1, v, p);
        p = list next(1, p):
    list_print(1, print_int);
    // Clean up the list, including payload
    list_kill(1);
    return 0;
```



18

21

27

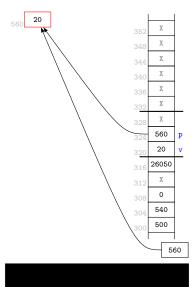
28

31

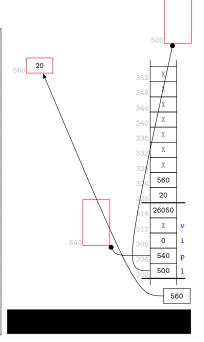
33 34

35

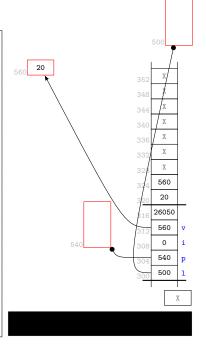
```
int *int_create(int v)
10
11
         int *p = malloc(sizeof(*p)); // allocate payload
12
        *p = v; // assign value
13
         return p;
14
15
     void int kill(void *v)
16
17
         free(v); // deallocate payload
19
     int main(void)
20
     {
         // Create empty list, hand over responsibility to
22
         // deallocate payload using int kill
23
         list *l = list_empty(int_kill);
24
         list_pos p = list_first(1);
25
         for (int i = 0; i < 2; i++) {
26
             int *v = int_create(20 + i);
             p = list_insert(1, v, p);
             p = list next(1, p):
29
30
         list_print(1, print_int);
32
36
37
         // Clean up the list, including payload
         list_kill(1);
39
         return 0;
40
```



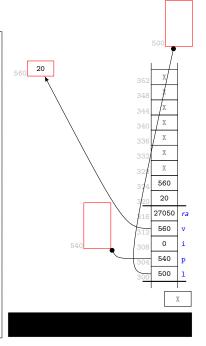
```
int *int_create(int v)
10
11
         int *p = malloc(sizeof(*p)); // allocate payload
12
        *p = v; // assign value
13
         return p;
14
15
     void int_kill(void *v)
16
17
         free(v); // deallocate payload
18
19
     int main(void)
20
     {
21
         // Create empty list, hand over responsibility to
22
         // deallocate payload using int kill
23
         list *l = list_empty(int_kill);
24
         list_pos p = list_first(1);
25
        for (int i = 0; i < 2; i++) {
26
             int *v = int_create(20 + i);
27
             p = list_insert(1, v, p);
28
             p = list next(1, p):
29
30
         list_print(1, print_int);
31
32
33
34
35
36
37
         // Clean up the list, including payload
38
         list_kill(1);
39
         return 0;
40
```



```
int *int_create(int v)
10
11
         int *p = malloc(sizeof(*p)); // allocate payload
12
        *p = v; // assign value
13
         return p;
14
15
     void int_kill(void *v)
16
17
         free(v); // deallocate payload
18
19
     int main(void)
20
     {
21
        // Create empty list, hand over responsibility to
22
         // deallocate payload using int kill
23
         list *l = list_empty(int_kill);
24
         list_pos p = list_first(1);
25
        for (int i = 0; i < 2; i++) {
26
             int *v = int_create(20 + i);
27
             p = list_insert(1, v, p);
28
             p = list next(1, p):
29
30
         list_print(1, print_int);
31
32
33
34
35
36
37
         // Clean up the list, including payload
38
         list_kill(1);
39
         return 0;
40
```



```
int *int_create(int v)
10
11
         int *p = malloc(sizeof(*p)); // allocate payload
12
        *p = v; // assign value
13
         return p;
14
15
     void int_kill(void *v)
16
17
         free(v); // deallocate payload
18
19
     int main(void)
20
     {
21
         // Create empty list, hand over responsibility to
22
         // deallocate payload using int kill
23
         list *l = list_empty(int_kill);
24
         list_pos p = list_first(1);
25
        for (int i = 0; i < 2; i++) {
26
             int *v = int_create(20 + i);
27
             p = list_insert(1, v, p);
28
             p = list next(1, p):
29
30
         list_print(1, print_int);
31
32
33
34
35
36
37
         // Clean up the list, including payload
38
         list_kill(1);
39
         return 0;
40
```



18

21

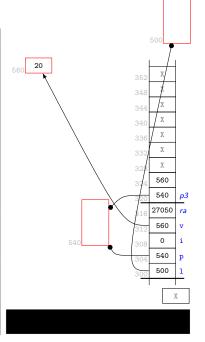
27

28

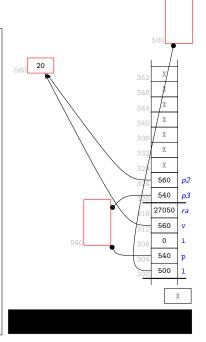
31

33 34

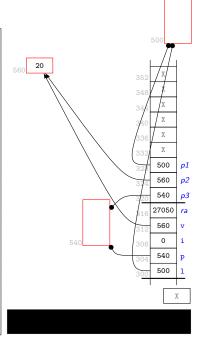
```
int *int_create(int v)
10
11
         int *p = malloc(sizeof(*p)); // allocate payload
12
        *p = v; // assign value
13
         return p;
14
15
     void int_kill(void *v)
16
17
         free(v); // deallocate payload
19
     int main(void)
20
     {
         // Create empty list, hand over responsibility to
22
         // deallocate payload using int kill
23
         list *l = list_empty(int_kill);
24
         list_pos p = list_first(1);
25
        for (int i = 0; i < 2; i++) {
26
             int *v = int_create(20 + i);
             p = list_insert(1, v, p);
             p = list next(1, p):
29
30
         list_print(1, print_int);
32
35
36
37
         // Clean up the list, including payload
         list_kill(1);
39
         return 0;
40
```



```
int *int_create(int v)
10
11
         int *p = malloc(sizeof(*p)); // allocate payload
12
        *p = v; // assign value
13
         return p;
14
15
     void int_kill(void *v)
16
17
         free(v); // deallocate payload
18
19
     int main(void)
20
     {
21
         // Create empty list, hand over responsibility to
22
         // deallocate payload using int kill
23
         list *l = list_empty(int_kill);
24
         list_pos p = list_first(1);
25
        for (int i = 0; i < 2; i++) {
26
             int *v = int_create(20 + i);
27
             p = list_insert(1, v, p);
28
             p = list next(1, p):
29
30
         list_print(1, print_int);
31
32
33
34
35
36
37
         // Clean up the list, including payload
38
         list_kill(1);
39
         return 0;
40
```



```
int *int_create(int v)
10
11
         int *p = malloc(sizeof(*p)); // allocate payload
12
        *p = v; // assign value
13
         return p;
14
15
     void int_kill(void *v)
16
17
         free(v); // deallocate payload
18
19
     int main(void)
20
     {
21
         // Create empty list, hand over responsibility to
22
         // deallocate payload using int kill
23
         list *l = list_empty(int_kill);
24
         list_pos p = list_first(1);
25
        for (int i = 0; i < 2; i++) {
26
             int *v = int_create(20 + i);
27
             p = list_insert(1, v, p);
28
             p = list next(1, p):
29
30
         list_print(1, print_int);
31
32
33
34
35
36
37
         // Clean up the list, including payload
38
         list_kill(1);
39
         return 0;
40
```



152

153

154

155

156

157

158

159

160

161

162

163

164 165

166

167

168

169

170

171

172

173

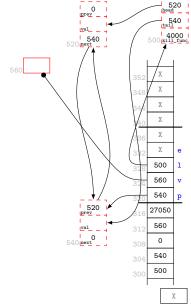
174

175

176

177

```
* list insert() - Insert a new element with a given value
 * Ql: List to manipulate.
 * Ov: Value (pointer) to be inserted into the list.
 * Op: 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 *1, 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:
```



152

153

154

155

156

157

158

159

160

161

162

163

164 165

166

167

168

169

170

171

172

173

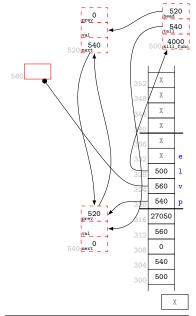
174

175

176

177

```
* list insert() - Insert a new element with a given value
 * Ql: List to manipulate.
 * Ov: Value (pointer) to be inserted into the list.
 * Op: 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 *1, 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:
```



152

153

154

155

156

157

158

159

160

161

162

163

164

165

166

167

168

169

170

171

172

173

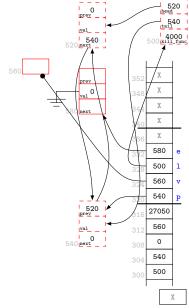
174

175

176

177

```
* list insert() - Insert a new element with a given value
 * Ql: List to manipulate.
 * Ov: Value (pointer) to be inserted into the list.
 * Op: 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 *1, 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:
```



152

153

154

155

156

157

158

159

160

161

162

163

164

165

166

167

168

169

170

171

172

173

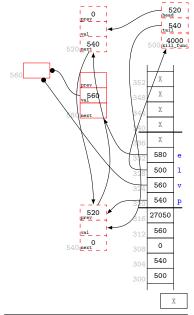
174

175

176

177

```
* list insert() - Insert a new element with a given value
 * Ql: List to manipulate.
 * Ov: Value (pointer) to be inserted into the list.
 * Op: 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 *1, 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:
```



152

153

154

155

156

157

158

159

160

161

162

163

164

165

166

167

168

169

170

171

172

173

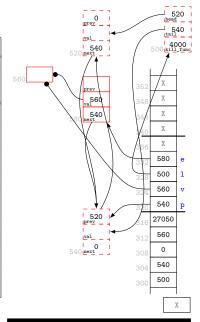
174

175

176

177

```
* list insert() - Insert a new element with a given value
 * Ql: List to manipulate.
 * Ov: Value (pointer) to be inserted into the list.
 * Op: 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 *1, 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:
```



152

153

154

155

156

157

158

159

160

161

162

163

164

165

166

167

168

169

170

171

172

173

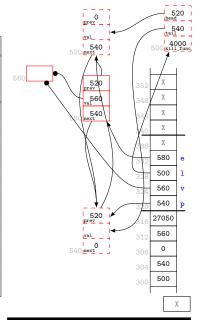
174

175

176

177

```
* list insert() - Insert a new element with a given value
 * Ql: List to manipulate.
 * Ov: Value (pointer) to be inserted into the list.
 * Op: 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 *1, 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:
```



152

153

154

155

156

157

158

159

160

161

162

163

164

165

166

167

168

169

170

171

172

173

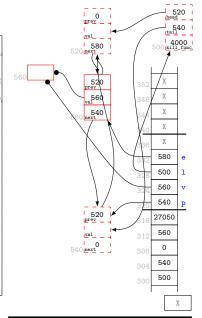
174

175

176

177

```
* list insert() - Insert a new element with a given value
 * Ql: List to manipulate.
 * Ov: Value (pointer) to be inserted into the list.
 * Op: 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 *1, 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:
```



152

153

154

155

156

157

158

159

160

161

162

163

164

165

166

167

168

169

170

171

172

173

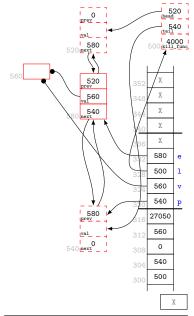
174

175

176

177

```
* list insert() - Insert a new element with a given value
 * Ql: List to manipulate.
 * Ov: Value (pointer) to be inserted into the list.
 * Op: 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 *1, 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:
```



152

153

154

155

156

157

158

159

160

161

162

163

164

165

166

167

168

169

170

171

172

173

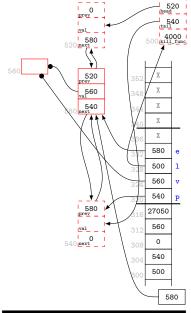
174

175

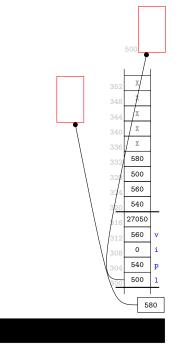
176

177

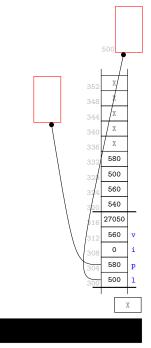
```
* list insert() - Insert a new element with a given value
 * Ql: List to manipulate.
 * Ov: Value (pointer) to be inserted into the list.
 * Op: Position in the list before which the value should
 * Creates a new element and inserts it into the list bef
 * Stores data in the new element.
 * Returns: The position of the newly created element.
 */
list pos list insert(list *1, 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:
```



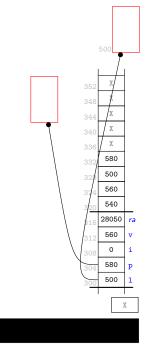
```
int *int_create(int v)
10
11
         int *p = malloc(sizeof(*p)); // allocate payload
12
        *p = v; // assign value
13
         return p;
14
15
     void int_kill(void *v)
16
17
         free(v); // deallocate payload
18
19
     int main(void)
20
     {
21
        // Create empty list, hand over responsibility to
22
        // deallocate payload using int kill
23
         list *l = list_empty(int_kill);
24
         list_pos p = list_first(1);
25
        for (int i = 0; i < 2; i++) {
26
             int *v = int_create(20 + i);
27
             p = list_insert(1, v, p);
28
             p = list next(1, p):
29
30
         list_print(1, print_int);
31
32
33
34
35
36
37
         // Clean up the list, including payload
38
         list_kill(1);
39
         return 0;
40
```



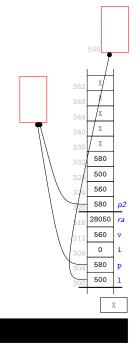
```
int *int_create(int v)
10
11
         int *p = malloc(sizeof(*p)); // allocate payload
12
        *p = v; // assign value
13
         return p;
14
15
     void int_kill(void *v)
16
17
         free(v); // deallocate payload
18
19
     int main(void)
20
     {
21
        // Create empty list, hand over responsibility to
22
        // deallocate payload using int kill
23
         list *l = list_empty(int_kill);
24
         list_pos p = list_first(1);
25
         for (int i = 0; i < 2; i++) {
26
             int *v = int_create(20 + i);
27
             p = list_insert(1, v, p);
28
             p = list_next(l, p);
29
30
         list_print(1, print_int);
31
32
33
34
35
36
37
         // Clean up the list, including payload
38
         list_kill(1);
39
         return 0;
40
```



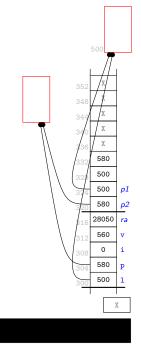
```
int *int_create(int v)
10
11
         int *p = malloc(sizeof(*p)); // allocate payload
12
        *p = v; // assign value
13
         return p;
14
15
     void int_kill(void *v)
16
17
         free(v); // deallocate payload
18
19
     int main(void)
20
     {
21
        // Create empty list, hand over responsibility to
22
        // deallocate payload using int kill
23
         list *l = list_empty(int_kill);
24
         list_pos p = list_first(1);
25
         for (int i = 0; i < 2; i++) {
26
             int *v = int_create(20 + i);
27
             p = list_insert(1, v, p);
28
             p = list_next(l, p);
29
30
         list_print(1, print_int);
31
32
33
34
35
36
37
         // Clean up the list, including payload
38
         list_kill(1);
39
         return 0;
40
```



```
int *int_create(int v)
10
11
         int *p = malloc(sizeof(*p)); // allocate payload
12
        *p = v; // assign value
13
         return p;
14
15
     void int_kill(void *v)
16
17
         free(v); // deallocate payload
18
19
     int main(void)
20
     {
21
        // Create empty list, hand over responsibility to
22
        // deallocate payload using int kill
23
         list *l = list_empty(int_kill);
24
         list_pos p = list_first(1);
25
         for (int i = 0; i < 2; i++) {
26
             int *v = int_create(20 + i);
27
             p = list_insert(1, v, p);
28
             p = list_next(l, p);
29
30
         list_print(1, print_int);
31
32
33
34
35
36
37
         // Clean up the list, including payload
38
         list_kill(1);
39
         return 0;
40
```



```
int *int_create(int v)
10
11
         int *p = malloc(sizeof(*p)); // allocate payload
12
        *p = v; // assign value
13
         return p;
14
15
     void int_kill(void *v)
16
17
         free(v); // deallocate payload
18
19
     int main(void)
20
     {
21
        // Create empty list, hand over responsibility to
22
        // deallocate payload using int kill
23
         list *l = list_empty(int_kill);
24
         list_pos p = list_first(1);
25
         for (int i = 0; i < 2; i++) {
26
             int *v = int_create(20 + i);
27
             p = list_insert(1, v, p);
28
             p = list_next(l, p);
29
30
         list_print(1, print_int);
31
32
33
34
35
36
37
         // Clean up the list, including payload
38
         list_kill(1);
39
         return 0;
40
```



112

113

114

115

116

117

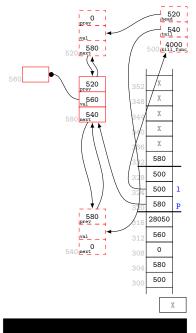
118

119

120

121 122

```
/**
 * list next() - Return the next position in a list.
 * @l: List to inspect.
 * Op: 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 *1, const list_pos p)
    return p->next;
```



112

113

114

115

116

117

118

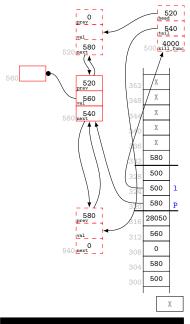
119

120

121

122

```
/**
 * list next() - Return the next position in a list.
 * @l: List to inspect.
 * Op: 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 *1, const list_pos p)
    return p->next;
```



112

113

114

115

116

117

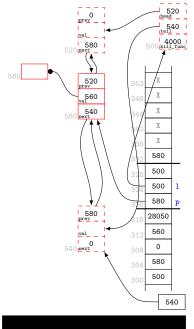
118

119

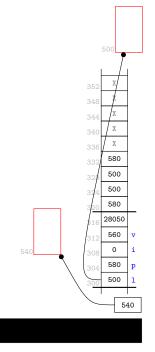
120 121

122

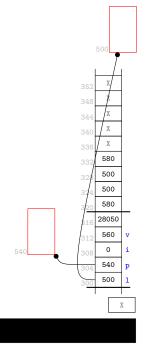
```
* list next() - Return the next position in a list.
 * @l: List to inspect.
 * Op: Any valid position except the last in the list.
 * Returns: The position in the list after the given posi
            NOTE: The return value is undefined for the l
list_pos list_next(const list *1, const list_pos p)
{
    return p->next;
```



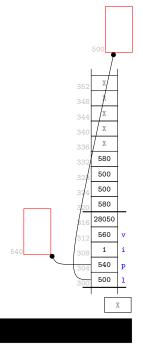
```
int *int_create(int v)
10
11
         int *p = malloc(sizeof(*p)); // allocate payload
12
        *p = v; // assign value
13
         return p;
14
15
     void int_kill(void *v)
16
17
         free(v); // deallocate payload
18
19
     int main(void)
20
     {
21
        // Create empty list, hand over responsibility to
22
        // deallocate payload using int kill
23
         list *l = list_empty(int_kill);
24
         list_pos p = list_first(1);
25
         for (int i = 0; i < 2; i++) {
26
             int *v = int_create(20 + i);
27
             p = list_insert(1, v, p);
28
             p = list_next(l, p);
29
30
         list_print(1, print_int);
31
32
33
34
35
36
37
         // Clean up the list, including payload
38
         list_kill(1);
39
         return 0;
40
```



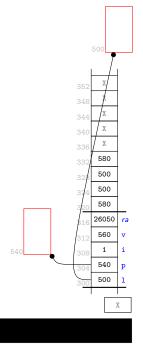
```
int *int_create(int v)
10
11
         int *p = malloc(sizeof(*p)); // allocate payload
12
        *p = v; // assign value
13
         return p;
14
15
     void int_kill(void *v)
16
17
         free(v); // deallocate payload
18
19
     int main(void)
20
     {
21
        // Create empty list, hand over responsibility to
22
         // deallocate payload using int kill
23
         list *l = list_empty(int_kill);
24
         list_pos p = list_first(1);
25
         for (int i = 0; i < 2; i++) {
26
             int *v = int_create(20 + i);
27
             p = list_insert(1, v, p);
28
             p = list next(1, p):
29
30
         list_print(1, print_int);
31
32
33
34
35
36
37
         // Clean up the list, including payload
38
         list_kill(1);
39
         return 0;
40
```



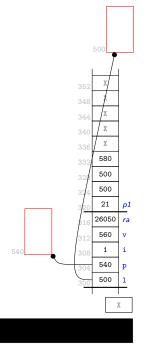
```
int *int_create(int v)
10
11
         int *p = malloc(sizeof(*p)); // allocate payload
12
        *p = v; // assign value
13
         return p;
14
15
     void int_kill(void *v)
16
17
         free(v); // deallocate payload
18
19
     int main(void)
20
     {
21
         // Create empty list, hand over responsibility to
22
         // deallocate payload using int kill
23
         list *l = list_empty(int_kill);
24
         list_pos p = list_first(1);
25
         for (int i = 0; i < 2; i++) {
26
             int *v = int_create(20 + i);
27
             p = list_insert(1, v, p);
28
             p = list next(1, p):
29
30
         list_print(1, print_int);
31
32
33
34
35
36
37
         // Clean up the list, including payload
38
         list_kill(1);
39
         return 0;
40
```



```
int *int_create(int v)
10
11
         int *p = malloc(sizeof(*p)); // allocate payload
12
        *p = v; // assign value
13
         return p;
14
15
     void int_kill(void *v)
16
17
         free(v); // deallocate payload
18
19
     int main(void)
20
     {
21
         // Create empty list, hand over responsibility to
22
         // deallocate payload using int kill
23
         list *l = list_empty(int_kill);
24
         list_pos p = list_first(1);
25
         for (int i = 0; i < 2; i++) {
26
             int *v = int_create(20 + i);
27
             p = list_insert(1, v, p);
28
             p = list next(1, p):
29
30
         list_print(1, print_int);
31
32
33
34
35
36
37
         // Clean up the list, including payload
38
         list_kill(1);
39
         return 0;
40
```



```
int *int_create(int v)
10
11
         int *p = malloc(sizeof(*p)); // allocate payload
12
        *p = v; // assign value
13
         return p;
14
15
     void int_kill(void *v)
16
17
         free(v); // deallocate payload
18
19
     int main(void)
20
     {
21
         // Create empty list, hand over responsibility to
22
         // deallocate payload using int kill
23
         list *l = list_empty(int_kill);
24
         list_pos p = list_first(1);
25
         for (int i = 0; i < 2; i++) {
26
             int *v = int_create(20 + i);
27
             p = list_insert(1, v, p);
28
             p = list next(1, p):
29
30
         list_print(1, print_int);
31
32
33
34
35
36
37
         // Clean up the list, including payload
38
         list_kill(1);
39
         return 0;
40
```



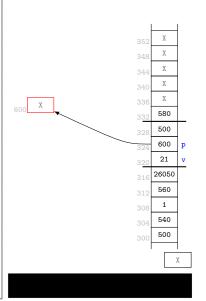
```
int *int_create(int v)
10
11
         int *p = malloc(sizeof(*p)); // allocate payload
12
         *p = v; // assign value
13
         return p;
14
15
     void int_kill(void *v)
16
17
         free(v); // deallocate payload
18
19
     int main(void)
20
     {
21
         // Create empty list, hand over responsibility to
22
         // deallocate payload using int kill
23
         list *l = list_empty(int_kill);
24
         list_pos p = list_first(1);
25
         for (int i = 0; i < 2; i++) {
26
             int *v = int_create(20 + i);
27
             p = list_insert(1, v, p);
28
             p = list next(1, p):
29
30
         list_print(1, print_int);
31
32
33
34
35
36
37
         // Clean up the list, including payload
38
         list_kill(1);
39
         return 0;
40
```

```
X
  X
  X
 580
 500
 500
 21
26050
 560
  1
 540
 500
```

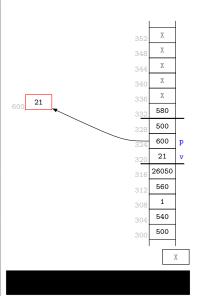
```
int *int_create(int v)
10
11
         int *p = malloc(sizeof(*p)); // allocate payload
12
         *p = v; // assign value
13
         return p;
14
    }
15
     void int_kill(void *v)
16
17
         free(v); // deallocate payload
18
19
     int main(void)
20
     {
21
         // Create empty list, hand over responsibility to
22
         // deallocate payload using int kill
23
         list *l = list_empty(int_kill);
24
         list_pos p = list_first(1);
25
         for (int i = 0; i < 2; i++) {
26
             int *v = int_create(20 + i);
27
             p = list_insert(1, v, p);
28
             p = list next(1, p):
29
30
         list_print(1, print_int);
31
32
33
34
35
36
37
         // Clean up the list, including payload
38
         list_kill(1);
39
         return 0;
40
```

```
X
  X
  X
 580
 500
 500
 21
26050
 560
  1
 540
 500
```

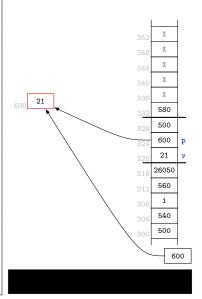
```
int *int_create(int v)
10
11
         int *p = malloc(sizeof(*p)); // allocate payload
12
         *p = v; // assign value
13
         return p;
14
15
     void int_kill(void *v)
16
17
         free(v); // deallocate payload
18
19
     int main(void)
20
     {
21
         // Create empty list, hand over responsibility to
         // deallocate payload using int_kill
22
23
         list *l = list_empty(int_kill);
24
         list_pos p = list_first(1);
25
         for (int i = 0; i < 2; i++) {
26
             int *v = int_create(20 + i);
27
             p = list_insert(1, v, p);
28
             p = list next(1, p):
29
30
         list_print(1, print_int);
31
32
33
34
35
36
37
         // Clean up the list, including payload
38
         list_kill(1);
39
         return 0;
40
```



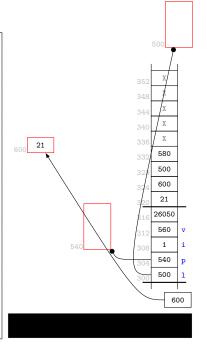
```
int *int_create(int v)
10
11
         int *p = malloc(sizeof(*p)); // allocate payload
12
         *p = v; // assign value
13
         return p;
14
15
     void int_kill(void *v)
16
17
         free(v); // deallocate payload
18
19
     int main(void)
20
     {
21
         // Create empty list, hand over responsibility to
         // deallocate payload using int_kill
22
23
         list *l = list_empty(int_kill);
24
         list_pos p = list_first(1);
25
         for (int i = 0; i < 2; i++) {
26
             int *v = int_create(20 + i);
27
             p = list_insert(1, v, p);
28
             p = list next(1, p):
29
30
         list_print(1, print_int);
31
32
33
34
35
36
37
         // Clean up the list, including payload
38
         list_kill(1);
39
         return 0;
40
```



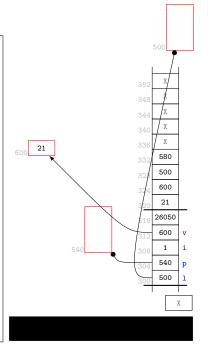
```
int *int_create(int v)
10
11
         int *p = malloc(sizeof(*p)); // allocate payload
12
        *p = v; // assign value
13
         return p;
14
15
     void int kill(void *v)
16
17
         free(v); // deallocate payload
18
19
     int main(void)
20
     {
21
         // Create empty list, hand over responsibility to
22
         // deallocate payload using int kill
23
         list *l = list_empty(int_kill);
24
         list_pos p = list_first(1);
25
         for (int i = 0; i < 2; i++) {
26
             int *v = int_create(20 + i);
27
             p = list_insert(1, v, p);
28
             p = list next(1, p):
29
30
         list_print(1, print_int);
31
32
33
34
35
36
37
         // Clean up the list, including payload
38
         list_kill(1);
39
         return 0;
40
```



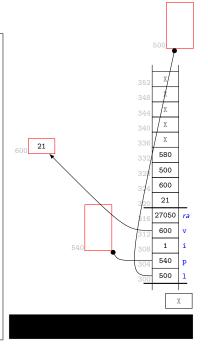
```
int *int_create(int v)
10
11
         int *p = malloc(sizeof(*p)); // allocate payload
12
        *p = v; // assign value
13
         return p;
14
15
     void int_kill(void *v)
16
17
         free(v); // deallocate payload
18
19
     int main(void)
20
     {
21
         // Create empty list, hand over responsibility to
22
         // deallocate payload using int kill
23
         list *l = list_empty(int_kill);
24
         list_pos p = list_first(1);
25
         for (int i = 0; i < 2; i++) {
26
             int *v = int_create(20 + i);
27
             p = list_insert(1, v, p);
28
             p = list next(1, p):
29
30
         list_print(1, print_int);
31
32
33
34
35
36
37
         // Clean up the list, including payload
38
         list_kill(1);
39
         return 0;
40
```



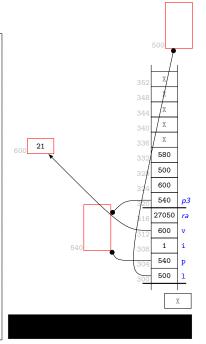
```
int *int_create(int v)
10
11
         int *p = malloc(sizeof(*p)); // allocate payload
12
        *p = v; // assign value
13
         return p;
14
15
     void int_kill(void *v)
16
17
         free(v); // deallocate payload
18
19
     int main(void)
20
     {
21
         // Create empty list, hand over responsibility to
22
         // deallocate payload using int kill
23
         list *l = list_empty(int_kill);
24
         list_pos p = list_first(1);
25
        for (int i = 0; i < 2; i++) {
26
             int *v = int_create(20 + i);
27
             p = list_insert(1, v, p);
28
             p = list next(1, p):
29
30
         list_print(1, print_int);
31
32
33
34
35
36
37
         // Clean up the list, including payload
38
         list_kill(1);
39
         return 0;
40
```



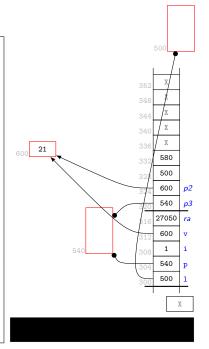
```
int *int_create(int v)
10
11
         int *p = malloc(sizeof(*p)); // allocate payload
12
        *p = v; // assign value
13
         return p;
14
15
     void int_kill(void *v)
16
17
         free(v); // deallocate payload
18
19
     int main(void)
20
     {
21
         // Create empty list, hand over responsibility to
22
         // deallocate payload using int kill
23
         list *l = list_empty(int_kill);
24
         list_pos p = list_first(1);
25
        for (int i = 0; i < 2; i++) {
26
             int *v = int_create(20 + i);
27
             p = list_insert(1, v, p);
28
             p = list next(1, p):
29
30
         list_print(1, print_int);
31
32
33
34
35
36
37
         // Clean up the list, including payload
38
         list_kill(1);
39
         return 0;
40
```



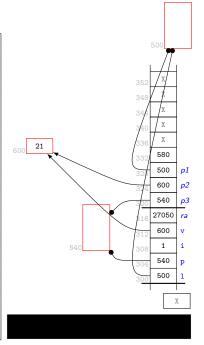
```
int *int_create(int v)
10
11
         int *p = malloc(sizeof(*p)); // allocate payload
12
        *p = v; // assign value
13
         return p;
14
15
     void int_kill(void *v)
16
17
         free(v); // deallocate payload
18
19
     int main(void)
20
     {
21
         // Create empty list, hand over responsibility to
22
         // deallocate payload using int kill
23
         list *l = list_empty(int_kill);
24
         list_pos p = list_first(1);
25
         for (int i = 0; i < 2; i++) {
26
             int *v = int_create(20 + i);
27
             p = list_insert(1, v, p);
28
             p = list next(1, p):
29
30
         list_print(1, print_int);
31
32
33
34
35
36
37
         // Clean up the list, including payload
38
         list_kill(1);
39
         return 0;
40
```



```
int *int_create(int v)
10
11
         int *p = malloc(sizeof(*p)); // allocate payload
12
        *p = v; // assign value
13
         return p;
14
15
     void int_kill(void *v)
16
17
         free(v); // deallocate payload
18
19
     int main(void)
20
     {
21
         // Create empty list, hand over responsibility to
22
         // deallocate payload using int kill
23
         list *l = list_empty(int_kill);
24
         list_pos p = list_first(1);
25
         for (int i = 0; i < 2; i++) {
26
             int *v = int_create(20 + i);
27
             p = list_insert(1, v, p);
28
             p = list next(1, p):
29
30
         list_print(1, print_int);
31
32
33
34
35
36
37
         // Clean up the list, including payload
38
         list_kill(1);
39
         return 0;
40
```



```
int *int_create(int v)
10
11
         int *p = malloc(sizeof(*p)); // allocate payload
12
        *p = v; // assign value
13
         return p;
14
15
     void int_kill(void *v)
16
17
         free(v); // deallocate payload
18
19
     int main(void)
20
     {
21
         // Create empty list, hand over responsibility to
22
         // deallocate payload using int kill
23
         list *l = list_empty(int_kill);
24
         list_pos p = list_first(1);
25
        for (int i = 0; i < 2; i++) {
26
             int *v = int_create(20 + i);
27
             p = list_insert(1, v, p);
28
             p = list next(1, p):
29
30
         list_print(1, print_int);
31
32
33
34
35
36
37
         // Clean up the list, including payload
38
         list_kill(1);
39
         return 0;
40
```



152

153

154

155

156

157

158

159

160

161

162

163

164

165

166

167

168

169

170

171

172

173

174

175

176

177

```
* list insert() - Insert a new element with a given value
 * Ql: List to manipulate.
 * Ov: Value (pointer) to be inserted into the list.
 * Op: 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 *1, 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:
```

```
0
   prev
                            540
                       4000
500 kill_func
     580
520 next
     520
                         X
     560
     540
                         X
                         X
                        580
                        500
                        600
                        540
     580
                       27050
   prev
                        600
                         1
540 next
                        540
                        500
```

152

153

154

155

156

157

158

159

160

161

162

163

164

165

166

167

168

169

170

171

172

173

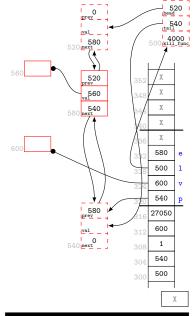
174

175

176

177

```
* list insert() - Insert a new element with a given value
 * Ql: List to manipulate.
 * Ov: Value (pointer) to be inserted into the list.
 * Op: 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 *1, 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:
```



152

153

154

155

156

157

158

159

160

161

162

163

164

165

166

167

168

169

170

171

172

173

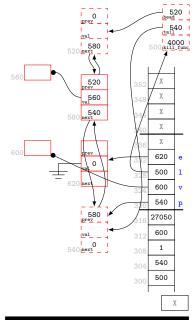
174

175

176

177

```
* list insert() - Insert a new element with a given value
 * Ql: List to manipulate.
 * Ov: Value (pointer) to be inserted into the list.
 * Op: 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 *1, 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:
```



152

153

154

155

156

157

158

159

160

161

162

163

164

165

166

167

168

169

170

171

172

173

174

175

176

177

```
* list insert() - Insert a new element with a given value
 * Ql: List to manipulate.
 * Ov: Value (pointer) to be inserted into the list.
 * Op: 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 *1, 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:
```

```
0
   prev
                            540
                       4000
500 kill_func
     580
520 next
     520
                         X
     560
     540
                         X
                         X
                        620
                        500
620 next
                        600
                        540
     580
                       27050
   prev
                        600
                         1
540 next
                        540
                        500
```

152

153

154

155

156

157

158

159

160

161

162

163

164

165

166

167

168

169

170

171

172

173

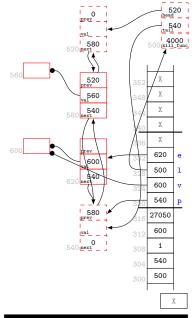
174

175

176

177

```
* list insert() - Insert a new element with a given value
 * Ql: List to manipulate.
 * Ov: Value (pointer) to be inserted into the list.
 * Op: 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 *1, 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:
```



152

153

154

155

156

157

158

159

160

161

162

163

164

165

166

167

168

169

170

171

172

173

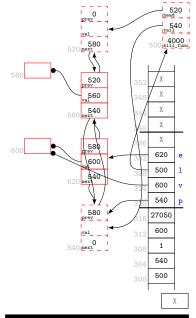
174

175

176

177

```
* list insert() - Insert a new element with a given value
 * Ql: List to manipulate.
 * Ov: Value (pointer) to be inserted into the list.
 * Op: 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 *1, 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:
```



152

153

154

155

156

157

158

159

160

161

162

163

164

165

166

167

168

169

170

171

172

173

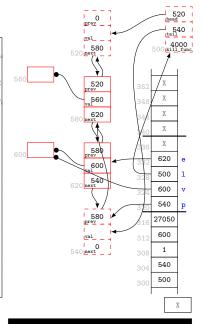
174

175

176

177

```
* list insert() - Insert a new element with a given value
 * Ql: List to manipulate.
 * Ov: Value (pointer) to be inserted into the list.
 * Op: 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 *1, 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:
```



152

153

154

155

156

157

158

159

160

161

162

163

164

165

166

167

168

169

170

171

172

173

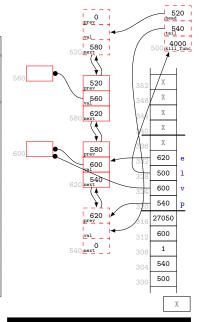
174

175

176

177

```
* list insert() - Insert a new element with a given value
 * Ql: List to manipulate.
 * Ov: Value (pointer) to be inserted into the list.
 * Op: 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 *1, 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:
```



152

153

154

155

156

157

158

159

160

161

162

163

164

165

166

167

168

169

170

171

172

173

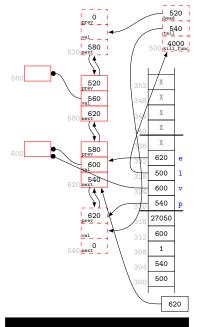
174

175

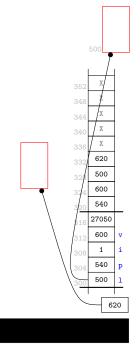
176

177

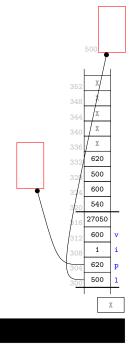
```
* list insert() - Insert a new element with a given value
 * Ql: List to manipulate.
 * Ov: Value (pointer) to be inserted into the list.
 * Op: Position in the list before which the value should
 * Creates a new element and inserts it into the list bef
 * Stores data in the new element.
 * Returns: The position of the newly created element.
 */
list pos list insert(list *1, 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:
```



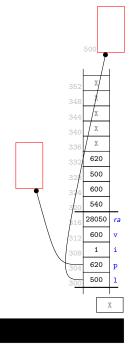
```
int *int_create(int v)
10
11
         int *p = malloc(sizeof(*p)); // allocate payload
12
        *p = v; // assign value
13
         return p;
14
15
     void int_kill(void *v)
16
17
         free(v); // deallocate payload
18
19
     int main(void)
20
     {
21
        // Create empty list, hand over responsibility to
22
         // deallocate payload using int kill
23
         list *l = list_empty(int_kill);
24
         list_pos p = list_first(1);
25
        for (int i = 0; i < 2; i++) {
26
             int *v = int_create(20 + i);
27
             p = list_insert(1, v, p);
28
             p = list next(1, p):
29
30
         list_print(1, print_int);
31
32
33
34
35
36
37
         // Clean up the list, including payload
38
         list_kill(1);
39
         return 0;
40
```



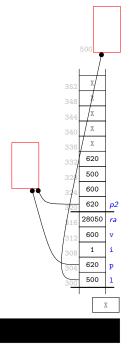
```
int *int_create(int v)
10
11
         int *p = malloc(sizeof(*p)); // allocate payload
12
        *p = v; // assign value
13
         return p;
14
15
     void int_kill(void *v)
16
17
         free(v); // deallocate payload
18
19
     int main(void)
20
     {
21
        // Create empty list, hand over responsibility to
22
        // deallocate payload using int kill
23
         list *l = list_empty(int_kill);
24
         list_pos p = list_first(1);
25
         for (int i = 0; i < 2; i++) {
26
             int *v = int_create(20 + i);
27
             p = list_insert(1, v, p);
28
             p = list_next(1, p);
29
30
         list_print(1, print_int);
31
32
33
34
35
36
37
         // Clean up the list, including payload
38
         list_kill(1);
39
         return 0;
40
```



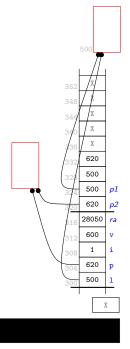
```
int *int_create(int v)
10
11
         int *p = malloc(sizeof(*p)); // allocate payload
12
        *p = v; // assign value
13
         return p;
14
15
     void int_kill(void *v)
16
17
         free(v); // deallocate payload
18
19
     int main(void)
20
     {
21
        // Create empty list, hand over responsibility to
22
         // deallocate payload using int kill
23
         list *l = list_empty(int_kill);
24
         list_pos p = list_first(1);
25
         for (int i = 0; i < 2; i++) {
26
             int *v = int_create(20 + i);
27
             p = list_insert(1, v, p);
28
             p = list_next(1, p);
29
30
         list_print(1, print_int);
31
32
33
34
35
36
37
         // Clean up the list, including payload
38
         list_kill(1);
39
         return 0;
40
```



```
int *int_create(int v)
10
11
         int *p = malloc(sizeof(*p)); // allocate payload
12
        *p = v; // assign value
13
         return p;
14
15
     void int_kill(void *v)
16
17
         free(v); // deallocate payload
18
19
     int main(void)
20
     {
21
         // Create empty list, hand over responsibility to
22
         // deallocate payload using int kill
23
         list *l = list_empty(int_kill);
24
         list_pos p = list_first(1);
25
         for (int i = 0; i < 2; i++) {
26
             int *v = int_create(20 + i);
27
             p = list_insert(1, v, p);
28
             p = list_next(1, p);
29
30
         list_print(1, print_int);
31
32
33
34
35
36
37
         // Clean up the list, including payload
38
         list_kill(1);
39
         return 0;
40
```



```
int *int_create(int v)
10
11
         int *p = malloc(sizeof(*p)); // allocate payload
12
        *p = v; // assign value
13
         return p;
14
15
     void int_kill(void *v)
16
17
         free(v); // deallocate payload
18
19
     int main(void)
20
     {
21
        // Create empty list, hand over responsibility to
22
        // deallocate payload using int kill
23
         list *l = list_empty(int_kill);
24
         list_pos p = list_first(1);
25
         for (int i = 0; i < 2; i++) {
26
             int *v = int_create(20 + i);
27
             p = list_insert(1, v, p);
28
             p = list_next(1, p);
29
30
         list_print(1, print_int);
31
32
33
34
35
36
37
         // Clean up the list, including payload
38
         list_kill(1);
39
         return 0;
40
```



112

113

114

115

116

117

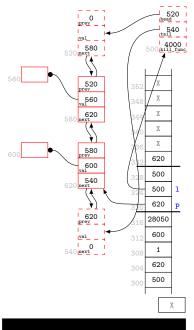
118

119

120

121 122

```
/**
 * list next() - Return the next position in a list.
 * @l: List to inspect.
 * Op: 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 *1, const list_pos p)
    return p->next;
```



112

113

114

115

116

117

 $\frac{118}{119}$

120

121

122

```
/**

* list_next() - Return the next position in a list.

* @l: List to inspect.

* @p: Any valid position except the last in the list.

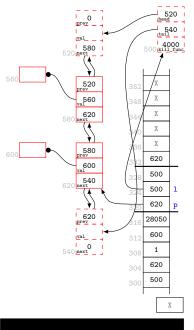
* Returns: The position in the list after the given posi

* NOTE: The return value is undefined for the l

*/

list_pos list_next(const list *1, const list_pos p)

{
    return p->next;
}
```



```
/**

* list_next() - Return the next position in a list.

* @l: List to inspect.

* @p: Any valid position except the last in the list.

*

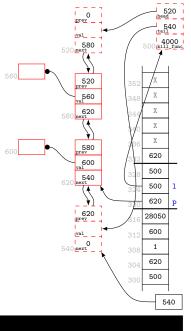
* Returns: The position in the list after the given posi

* NOTE: The return value is undefined for the l

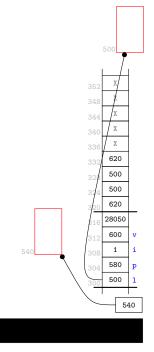
*/

list_pos list_next(const list *l, const list_pos p)

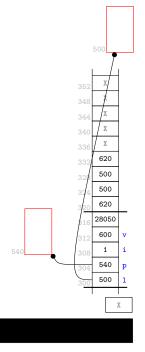
{
    return p->next;
}
```



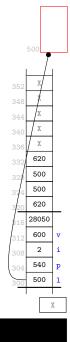
```
int *int_create(int v)
10
11
         int *p = malloc(sizeof(*p)); // allocate payload
12
        *p = v; // assign value
13
         return p;
14
15
     void int_kill(void *v)
16
17
         free(v); // deallocate payload
18
19
     int main(void)
20
     {
21
        // Create empty list, hand over responsibility to
22
         // deallocate payload using int kill
23
         list *l = list_empty(int_kill);
24
         list_pos p = list_first(1);
25
         for (int i = 0; i < 2; i++) {
26
             int *v = int_create(20 + i);
27
             p = list_insert(1, v, p);
28
             p = list_next(1, p);
29
30
         list_print(1, print_int);
31
32
33
34
35
36
37
         // Clean up the list, including payload
38
         list_kill(1);
39
         return 0;
40
```



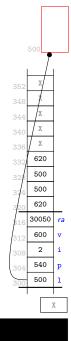
```
int *int_create(int v)
10
11
         int *p = malloc(sizeof(*p)); // allocate payload
12
        *p = v; // assign value
13
         return p;
14
15
     void int_kill(void *v)
16
17
         free(v); // deallocate payload
18
19
     int main(void)
20
     {
21
        // Create empty list, hand over responsibility to
22
         // deallocate payload using int kill
23
         list *l = list_empty(int_kill);
24
         list_pos p = list_first(1);
25
         for (int i = 0; i < 2; i++) {
26
             int *v = int_create(20 + i);
27
             p = list_insert(1, v, p);
28
             p = list next(1, p):
29
30
         list_print(1, print_int);
31
32
33
34
35
36
37
         // Clean up the list, including payload
38
         list_kill(1);
39
         return 0;
40
```



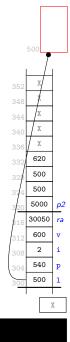
```
int *int_create(int v)
10
11
         int *p = malloc(sizeof(*p)); // allocate payload
12
        *p = v; // assign value
13
         return p;
14
15
     void int_kill(void *v)
16
17
         free(v); // deallocate payload
18
19
     int main(void)
20
     {
21
        // Create empty list, hand over responsibility to
22
        // deallocate payload using int kill
23
         list *l = list_empty(int_kill);
24
         list_pos p = list_first(1);
25
        for (int i = 0; i < 2; i++) {
26
             int *v = int_create(20 + i);
27
             p = list_insert(1, v, p);
28
             p = list next(1, p):
29
        list print(1, print_int);
30
31
32
33
34
35
36
37
         // Clean up the list, including payload
38
         list_kill(1);
39
         return 0;
40
```



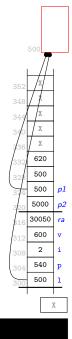
```
int *int_create(int v)
10
11
         int *p = malloc(sizeof(*p)); // allocate payload
12
        *p = v; // assign value
13
         return p;
14
15
     void int_kill(void *v)
16
17
         free(v); // deallocate payload
18
19
     int main(void)
20
     {
21
        // Create empty list, hand over responsibility to
22
        // deallocate payload using int kill
23
         list *l = list_empty(int_kill);
24
         list_pos p = list_first(1);
25
        for (int i = 0; i < 2; i++) {
26
             int *v = int_create(20 + i);
27
             p = list_insert(1, v, p);
28
             p = list next(1, p):
29
        list print(1, print_int);
30
31
32
33
34
35
36
37
         // Clean up the list, including payload
38
         list_kill(1);
39
         return 0;
40
```



```
int *int_create(int v)
10
11
         int *p = malloc(sizeof(*p)); // allocate payload
12
        *p = v; // assign value
13
         return p;
14
15
     void int_kill(void *v)
16
17
         free(v); // deallocate payload
18
19
     int main(void)
20
     {
21
        // Create empty list, hand over responsibility to
22
        // deallocate payload using int kill
23
         list *l = list_empty(int_kill);
24
         list_pos p = list_first(1);
25
        for (int i = 0; i < 2; i++) {
26
             int *v = int_create(20 + i);
27
             p = list_insert(1, v, p);
28
             p = list next(1, p):
29
        list print(1, print_int);
30
31
32
33
34
35
36
37
         // Clean up the list, including payload
38
         list_kill(1);
39
         return 0;
40
```



```
int *int_create(int v)
10
11
         int *p = malloc(sizeof(*p)); // allocate payload
12
        *p = v; // assign value
13
         return p;
14
15
     void int_kill(void *v)
16
17
         free(v); // deallocate payload
18
19
     int main(void)
20
     {
21
        // Create empty list, hand over responsibility to
22
        // deallocate payload using int kill
23
         list *l = list_empty(int_kill);
24
         list_pos p = list_first(1);
25
        for (int i = 0; i < 2; i++) {
26
             int *v = int_create(20 + i);
27
             p = list_insert(1, v, p);
28
             p = list next(1, p):
29
        list print(1, print_int);
30
31
32
33
34
35
36
37
         // Clean up the list, including payload
38
         list_kill(1);
39
         return 0;
40
```



239

240 241

242

243

244

245

246

247

248

249

250

251

252

253

254

255

256

257

258

259

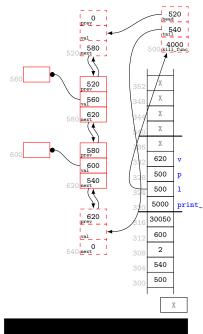
260

261

262

263 264

```
/**
* list print() - Iterate over the list element and print
* @l: List to inspect.
* Oprint 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 *1, inspect_callback print_fun
   // Start at the beginning of the list.
   list_pos p = list_first(1);
   printf("( ");
   while (!list_pos_is_equal(1, p, list_end(1))) {
        // 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(1, p);
   printf(" )\n"):
```



239

240 241

242

243

244

245

246

247

248

249

250

251

252

253

254

255

256

257

258

259

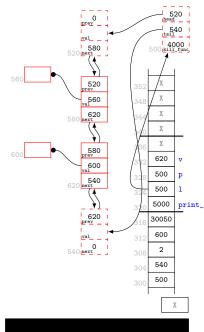
260

261

262

263 264

```
/**
 * list print() - Iterate over the list element and print
 * @l: List to inspect.
 * Oprint 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 *1, inspect_callback print_fun
{
    // Start at the beginning of the list.
    list_pos p = list_first(1);
    printf("( ");
    while (!list_pos_is_equal(1, p, list_end(1))) {
        // Call print func with the element value at the
        // current position.
        void *v=list_inspect(1, p);
        print func(v):
        // Advance to next position.
        p = list_next(1, p);
    printf(" )\n"):
```



239

240 241

242

243

244

245

246

247

248

249

250

251

252

253

254

255

256

257

258

259

260

261

262

263 264

```
/**
 * list print() - Iterate over the list element and print
 * @l: List to inspect.
 * Oprint 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 *1, inspect_callback print_fun
{
   // Start at the beginning of the list.
    list_pos p = list_first(1);
    printf("( ");
    while (!list pos is equal(1, p, list end(1))) {
        // Call print func with the element value at the
       // current position.
       void *v=list_inspect(1, p);
       print func(v):
       // Advance to next position.
       p = list_next(1, p);
    printf(" )\n"):
```

```
0
   prev
                          540
tail
                       4000
500 kill_func
     580
520 next
     520
                         X
     560
     620
                         X
                         X
     580
                        620
     600
                        580
     540
620 next
                        500
                       5000
                              print
     620
                       30050
   prev
                        600
      Ω
                         2
540 next
                        540
                        500
```

239

240 241

242

243

244

245

246

247

248

249

250

251

252

253

254

255

256

257

258

259

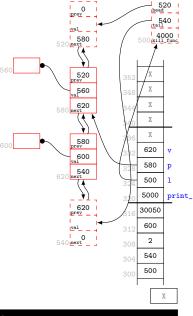
260

261

262

263 264

```
/**
 * list print() - Iterate over the list element and print
 * @l: List to inspect.
 * Oprint 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 *1, inspect_callback print_fun
{
    // Start at the beginning of the list.
    list_pos p = list_first(1);
    printf("( ");
    while (!list pos is equal(1, p, list end(1))) {
        // 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(1, p);
    printf(" )\n"):
```



239

 $\frac{240}{241}$

 $\frac{242}{243}$

244

245

246

247

248

249

250

251

252

253

254

255

256

257

258

259

260

261

262

 $\frac{263}{264}$

```
/**
 * list print() - Iterate over the list element and print
 * @l: List to inspect.
 * Oprint 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 *1, inspect_callback print_fun
{
   // Start at the beginning of the list.
    list_pos p = list_first(1);
    printf("( ");
    while (!list_pos_is_equal(1, p, list_end(1))) {
        // 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(1, p);
    printf(" )\n"):
```

```
0
   prev
                         540
                     4000
500 kill_func
     580
520 next
     520
                        X
     560
     620
                        X
                        X
     580
                       620
     600
                       580
     540
620 next
                       500
                      5000
                             print
     620
                      30050
   prev
                       600
      Ω
                        2
540 next
                       540
                       500
```

239

 $\frac{240}{241}$

242

243

244

245

246

247

248

249

250

251

252

253

254

255

256

257

258

259

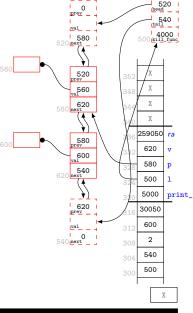
260

261

262

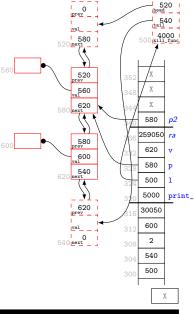
 $\frac{263}{264}$

```
/**
 * list print() - Iterate over the list element and print
 * @l: List to inspect.
 * Oprint 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 *1, inspect_callback print_fun
{
   // Start at the beginning of the list.
    list_pos p = list_first(1);
    printf("( ");
    while (!list_pos_is_equal(1, p, list_end(1))) {
        // 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(1, p);
    printf(" )\n"):
```



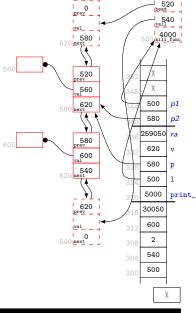
239

```
/**
240
241
       * list print() - Iterate over the list element and print
       * @l: List to inspect.
242
243
       * Oprint func: Function called for each element.
244
245
       * Iterates over the list and calls print func with the v
246
       * in each element.
247
248
       * Returns: Nothing.
249
       */
250
      void list_print(const list *1, inspect_callback print_fun
251
      {
252
         // Start at the beginning of the list.
253
          list_pos p = list_first(1);
254
255
          printf("( ");
256
          while (!list_pos_is_equal(1, p, list_end(1))) {
257
              // Call print func with the element value at the
258
              // current position.
259
              void *v=list_inspect(l, p);
260
             print func(v):
261
             // Advance to next position.
262
              p = list_next(1, p);
263
          printf(" )\n"):
265
```



239

```
/**
240
241
       * list print() - Iterate over the list element and print
       * @l: List to inspect.
242
243
       * Oprint func: Function called for each element.
244
245
       * Iterates over the list and calls print func with the v
246
       * in each element.
247
248
       * Returns: Nothing.
249
       */
250
      void list_print(const list *1, inspect_callback print_fun
251
      {
252
         // Start at the beginning of the list.
253
          list_pos p = list_first(1);
254
255
          printf("( ");
256
          while (!list_pos_is_equal(1, p, list_end(1))) {
257
              // Call print func with the element value at the
258
              // current position.
259
              void *v=list_inspect(l, p);
260
             print func(v):
261
             // Advance to next position.
262
              p = list_next(1, p);
263
          printf(" )\n"):
265
```



 $\frac{147}{148}$

```
/**

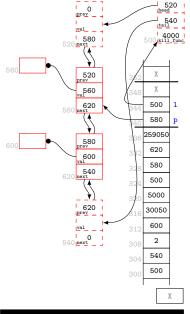
* list_inspect() - Return the value of the element at a position in a list.

* Cl: List to inspect.

* Op: 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 left wid * list_inspect(const list *1, const list_pos p)

{ return p->val;
}
```



```
/**

* list_inspect() - Return the value of the element at a position in a list.

* Ol: List to inspect.

* Op: Any valid position in the list, except the last.

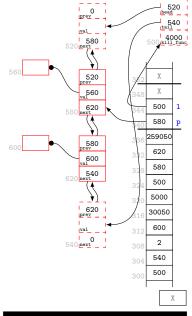
* Returns: Returns the value at the given position as a *

* NOTE: The return value is undefined for the last.

*/

void *list_inspect(const list *1, const list_pos p)

{
    return p->val;
}
```



 $\frac{147}{148}$

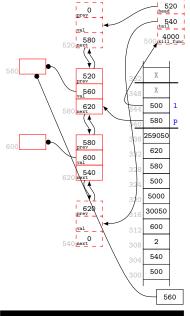
```
/**

* list_inspect() - Return the value of the element at a position in a list.

* Cl: List to inspect.

* Op: Any valid position in the list, except the last.

* Returns: Returns the value at the given position as a position
```



239

240 241

242 243

248

250

252

253

254

255

256

257

258

259

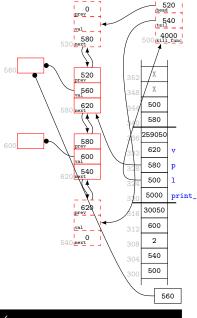
260

261

262

263 264

```
/**
       * list print() - Iterate over the list element and print
       * @l: List to inspect.
       * Oprint func: Function called for each element.
244
245
       * Iterates over the list and calls print func with the v
246
       * in each element.
247
       * Returns: Nothing.
249
       */
      void list_print(const list *1, inspect_callback print_fun
251
      {
         // Start at the beginning of the list.
          list_pos p = list_first(1);
          printf("( ");
          while (!list_pos_is_equal(1, p, list_end(1))) {
              // 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(1, p);
          printf(" )\n"):
```



239

 $\frac{240}{241}$

 $\frac{242}{243}$

244

245

246

247

248

249

250

251

252

253

254

255

256

257

258

259

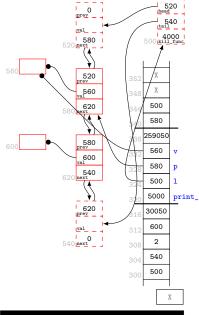
260

261

262

 $\frac{263}{264}$

```
/**
 * list print() - Iterate over the list element and print
 * @l: List to inspect.
 * Oprint 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 *1, inspect_callback print_fun
{
   // Start at the beginning of the list.
    list_pos p = list_first(1);
    printf("( ");
    while (!list_pos_is_equal(1, p, list_end(1))) {
        // 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(1, p);
    printf(" )\n"):
```



239

 $\frac{240}{241}$

 $\frac{242}{243}$

244

245

246

247

248

249

250

251

252

253

254

255

256

257

258

259

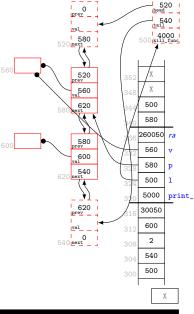
260

261

262

 $\frac{263}{264}$

```
/**
 * list print() - Iterate over the list element and print
 * @l: List to inspect.
 * Oprint 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 *1, inspect_callback print_fun
{
   // Start at the beginning of the list.
    list_pos p = list_first(1);
    printf("( ");
    while (!list_pos_is_equal(1, p, list_end(1))) {
        // 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(1, p);
    printf(" )\n"):
```



239

240 241

242 243

244

245

246

247

248

249

250

251

252

253

254

255

256

257

258

259

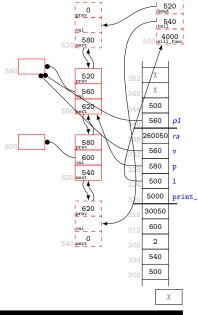
260

261

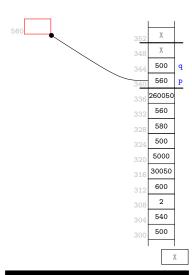
262

263 264

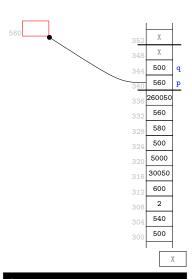
```
/**
 * list print() - Iterate over the list element and print
 * @l: List to inspect.
 * Oprint func: Function called for each element.
 * Iterates over the list and calls print func with the v
 * in each element.
 * Returns: Nothing.
 */
void list_print(const list *1, inspect_callback print_fun
{
   // Start at the beginning of the list.
    list_pos p = list_first(1);
    printf("( ");
    while (!list_pos_is_equal(1, p, list_end(1))) {
        // 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(1, p);
    printf(" )\n"):
```



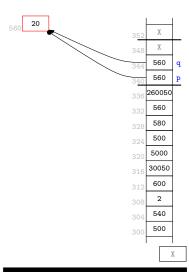
```
void print_int(const void *p)
{
    const int *q = p;
    printf("%d ", *q);
}
```



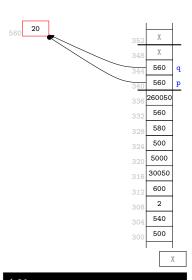
```
void print_int(const void *p)
{
    const int *q = p;
    printf("%d ", *q);
}
```



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



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



239

240 241

242 243

248

250

252

253

254

255

256

257

258

259

260

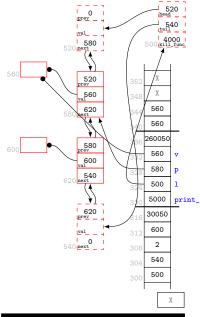
261

262

263 264

265

```
/**
       * list print() - Iterate over the list element and print
       * @l: List to inspect.
       * Oprint func: Function called for each element.
244
245
       * Iterates over the list and calls print func with the v
246
       * in each element.
247
       * Returns: Nothing.
249
       */
      void list_print(const list *1, inspect_callback print_fun
251
      {
         // Start at the beginning of the list.
          list_pos p = list_first(1);
          printf("( ");
          while (!list_pos_is_equal(1, p, list_end(1))) {
              // 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(1, p);
          printf(" )\n"):
```



239

 $\frac{240}{241}$

 $\frac{242}{243}$

244

245

246

247

248

249

250

251

252

253

254

255

256

257

258

259

260

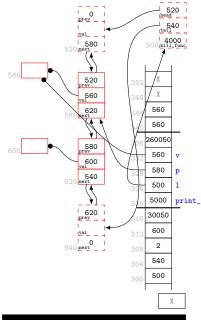
261

262

 $\frac{263}{264}$

265

```
/**
 * list print() - Iterate over the list element and print
 * @l: List to inspect.
 * Oprint 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 *1, inspect_callback print_fun
{
   // Start at the beginning of the list.
    list_pos p = list_first(1);
    printf("( ");
    while (!list_pos_is_equal(1, p, list_end(1))) {
        // 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(1, p);
    printf(" )\n"):
```



239

240 241

242

243

248

250

252

253

254

255

256

257

258

259

260

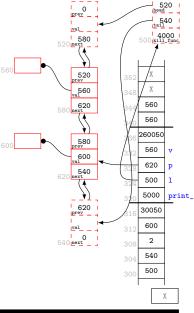
261

262

263 264

265

```
/**
       * list print() - Iterate over the list element and print
       * @l: List to inspect.
       * Oprint func: Function called for each element.
244
245
       * Iterates over the list and calls print func with the v
246
       * in each element.
247
       * Returns: Nothing.
249
       */
      void list_print(const list *1, inspect_callback print_fun
251
      {
          // Start at the beginning of the list.
          list_pos p = list_first(1);
          printf("( ");
          while (!list pos is equal(1, p, list end(1))) {
              // 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(1, p);
          printf(" )\n"):
```



239

240 241

242 243

248

250

252

253

254

255

256

257

258

259

260

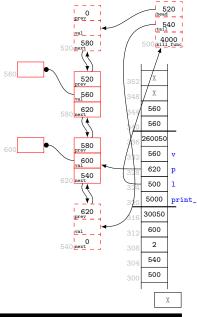
261

262

263 264

265

```
/**
       * list print() - Iterate over the list element and print
       * @l: List to inspect.
       * Oprint func: Function called for each element.
244
245
       * Iterates over the list and calls print func with the v
246
       * in each element.
247
       * Returns: Nothing.
249
       */
      void list_print(const list *1, inspect_callback print_fun
251
      {
         // Start at the beginning of the list.
          list_pos p = list_first(1);
          printf("( ");
          while (!list_pos_is_equal(1, p, list_end(1))) {
              // 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(1, p);
          printf(" )\n"):
```



239

240 241

242 243

248

250

252

253

254

255

256

257

258

259

260

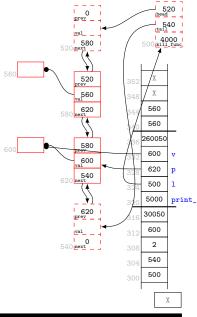
261

262

263 264

265

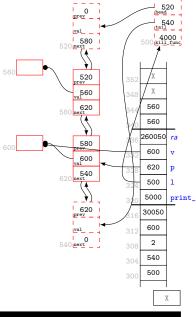
```
/**
       * list print() - Iterate over the list element and print
       * @l: List to inspect.
       * Oprint func: Function called for each element.
244
245
       * Iterates over the list and calls print func with the v
246
       * in each element.
247
       * Returns: Nothing.
249
       */
      void list_print(const list *1, inspect_callback print_fun
251
      {
         // Start at the beginning of the list.
          list_pos p = list_first(1);
          printf("( ");
          while (!list_pos_is_equal(1, p, list_end(1))) {
              // 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(1, p);
          printf(" )\n"):
```



239

264

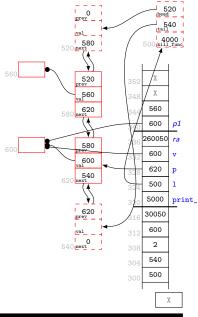
```
/**
240
241
       * list print() - Iterate over the list element and print
       * @l: List to inspect.
242
243
       * Oprint func: Function called for each element.
244
245
       * Iterates over the list and calls print func with the v
246
       * in each element.
247
248
       * Returns: Nothing.
249
       */
250
      void list_print(const list *1, inspect_callback print_fun
251
      {
252
         // Start at the beginning of the list.
253
          list_pos p = list_first(1);
254
255
          printf("( ");
256
          while (!list_pos_is_equal(1, p, list_end(1))) {
257
              // Call print func with the element value at the
258
             // current position.
259
              void *v=list_inspect(l, p);
260
             print func(v):
261
             // Advance to next position.
262
              p = list_next(1, p);
263
          printf(" )\n"):
265
```



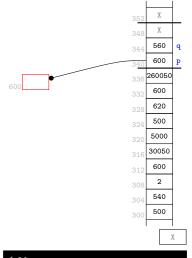
239

264

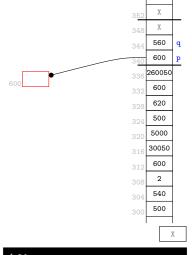
```
/**
240
241
       * list print() - Iterate over the list element and print
       * @l: List to inspect.
242
243
       * Oprint func: Function called for each element.
244
245
       * Iterates over the list and calls print func with the v
246
       * in each element.
247
248
       * Returns: Nothing.
249
       */
250
      void list_print(const list *1, inspect_callback print_fun
251
      {
252
         // Start at the beginning of the list.
253
          list_pos p = list_first(1);
254
255
          printf("( ");
256
          while (!list_pos_is_equal(1, p, list_end(1))) {
257
              // Call print func with the element value at the
258
             // current position.
259
              void *v=list_inspect(l, p);
260
             print func(v):
261
             // Advance to next position.
262
              p = list_next(1, p);
263
          printf(" )\n"):
265
```



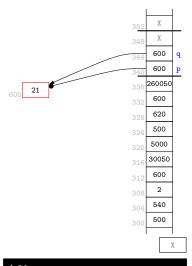
```
void print_int(const void *p)
{
    const int *q = p;
    printf("%d ", *q);
}
```



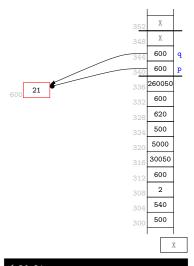
```
void print_int(const void *p)
{
    const int *q = p;
    printf("%d ", *q);
}
```



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



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



(20 21

239

240 241

242 243

248

250

252

253

254

255

256

257

258

259

260

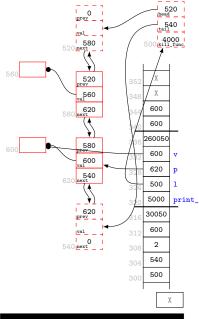
261

262

263 264

265

```
/**
       * list print() - Iterate over the list element and print
       * @l: List to inspect.
       * Oprint func: Function called for each element.
244
245
       * Iterates over the list and calls print func with the v
246
       * in each element.
247
       * Returns: Nothing.
249
       */
      void list_print(const list *1, inspect_callback print_fun
251
      {
         // Start at the beginning of the list.
          list_pos p = list_first(1);
          printf("( ");
          while (!list_pos_is_equal(1, p, list_end(1))) {
              // 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(1, p);
          printf(" )\n"):
```



239

 $\frac{240}{241}$

 $\frac{242}{243}$

244

245

246

247

248

249

250

251

252

253

254

255

256

257

258

259

260

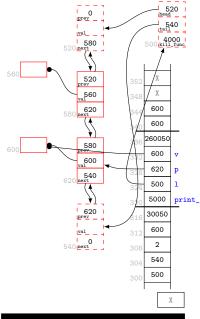
261

262

 $\frac{263}{264}$

265

```
/**
 * list print() - Iterate over the list element and print
 * @l: List to inspect.
 * Oprint 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 *1, inspect_callback print_fun
{
    // Start at the beginning of the list.
    list_pos p = list_first(1);
    printf("( ");
    while (!list_pos_is_equal(1, p, list_end(1))) {
        // 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(1, p);
    printf(" )\n"):
```



239

 $\frac{240}{241}$

 $\frac{242}{243}$

244

245

246

247

248

249

250

251

252

253

254

255

256

257

258

259

260

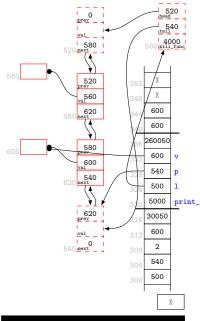
261

262

 $\frac{263}{264}$

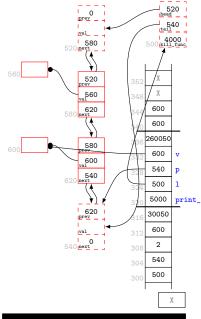
265

```
/**
 * list print() - Iterate over the list element and print
 * @l: List to inspect.
 * Oprint 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 *1, inspect_callback print_fun
{
    // Start at the beginning of the list.
    list_pos p = list_first(1);
    printf("( ");
    while (!list pos is equal(1, p, list end(1))) {
        // 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(1, p);
    printf(" )\n"):
```



239

```
/**
240
241
       * list print() - Iterate over the list element and print
       * @l: List to inspect.
242
243
       * Oprint func: Function called for each element.
244
245
       * Iterates over the list and calls print func with the v
246
       * in each element.
247
248
       * Returns: Nothing.
249
       */
250
      void list_print(const list *1, inspect_callback print_fun
251
      {
252
         // Start at the beginning of the list.
253
          list_pos p = list_first(1);
254
255
          printf("( ");
256
          while (!list_pos_is_equal(1, p, list_end(1))) {
257
              // Call print func with the element value at the
258
             // current position.
259
             void *v=list_inspect(1, p);
260
             print func(v):
261
             // Advance to next position.
262
              p = list_next(1, p);
263
264
          printf(" )\n");
265
```



239

 $\frac{240}{241}$

 $\frac{242}{243}$

244

245

246

247

248

249

250

251

252

253

254

255

256

257

258

259

260

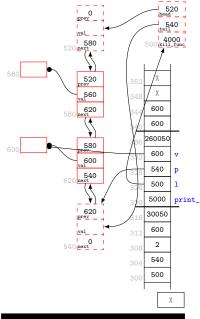
261

262

 $\frac{263}{264}$

265

```
/**
 * list print() - Iterate over the list element and print
 * @l: List to inspect.
 * Oprint 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 *1, inspect_callback print_fun
{
   // Start at the beginning of the list.
    list_pos p = list_first(1);
    printf("( ");
    while (!list_pos_is_equal(1, p, list_end(1))) {
        // Call print func with the element value at the
       // current position.
       void *v=list_inspect(1, p);
       print func(v):
       // Advance to next position.
       p = list_next(1, p);
    printf(" )\n");
```



```
int *int_create(int v)
10
11
         int *p = malloc(sizeof(*p)); // allocate payload
12
         *p = v; // assign value
13
         return p;
14
15
     void int_kill(void *v)
16
17
         free(v); // deallocate payload
18
19
     int main(void)
20
     {
21
        // Create empty list, hand over responsibility to
22
        // deallocate payload using int kill
23
         list *l = list_empty(int_kill);
24
         list_pos p = list_first(1);
25
        for (int i = 0; i < 2; i++) {
26
             int *v = int_create(20 + i);
27
             p = list_insert(1, v, p);
28
             p = list next(1, p):
29
        list print(1, print_int);
30
31
32
33
34
35
36
37
         // Clean up the list, including payload
38
         list_kill(1);
39
         return 0;
40
```

```
600
     1600
   260050
     600
     540
     500
32
     5000
    30050
     600
312
      2
     540
     500
```

```
int *int_create(int v)
10
         int *p = malloc(sizeof(*p)); // allocate payload
11
12
         *p = v; // assign value
13
         return p;
14
15
     void int_kill(void *v)
16
17
         free(v); // deallocate payload
18
19
     int main(void)
20
     {
21
         // Create empty list, hand over responsibility to
22
         // deallocate payload using int kill
23
         list *l = list_empty(int_kill);
24
         list_pos p = list_first(1);
25
         for (int i = 0; i < 2; i++) {
26
             int *v = int_create(20 + i);
27
             p = list_insert(1, v, p);
28
             p = list next(1, p):
29
30
         list_print(1, print_int);
31
32
33
34
35
36
37
         // Clean up the list, including payload
38
         list_kill(1);
39
         return 0;
40
```

```
600
     1600
   260050
     600
     540
     500
32
     5000
    30050
     600
312
      2
     540
     500
```

```
int *int_create(int v)
10
         int *p = malloc(sizeof(*p)); // allocate payload
11
12
         *p = v; // assign value
13
         return p;
14
15
     void int_kill(void *v)
16
17
         free(v); // deallocate payload
18
19
     int main(void)
20
     {
21
         // Create empty list, hand over responsibility to
22
         // deallocate payload using int kill
23
         list *l = list_empty(int_kill);
24
         list_pos p = list_first(1);
25
         for (int i = 0; i < 2; i++) {
26
             int *v = int_create(20 + i);
27
             p = list_insert(1, v, p);
28
             p = list next(1, p):
29
30
         list_print(1, print_int);
31
32
33
34
35
36
37
         // Clean up the list, including payload
38
         list_kill(1);
39
         return 0;
40
```

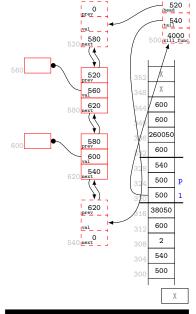
```
600
     1600
   260050
     600
     540
     500
32
    5000
    38050
     600
312
      2
     540
     500
```

```
int *int_create(int v)
10
         int *p = malloc(sizeof(*p)); // allocate payload
11
12
        *p = v; // assign value
13
         return p;
14
15
     void int_kill(void *v)
16
17
         free(v); // deallocate payload
18
19
     int main(void)
20
     {
21
         // Create empty list, hand over responsibility to
22
         // deallocate payload using int kill
23
         list *l = list_empty(int_kill);
24
         list_pos p = list_first(1);
25
         for (int i = 0; i < 2; i++) {
26
             int *v = int_create(20 + i);
27
             p = list_insert(1, v, p);
28
             p = list next(1, p):
29
30
         list_print(1, print_int);
31
32
33
34
35
36
37
         // Clean up the list, including payload
38
         list_kill(1);
39
         return 0;
40
```

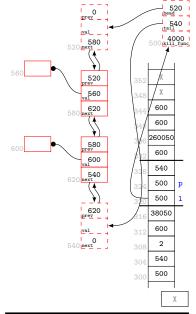
```
600
     1600
   260050
     600
     540
     500
     500
            р1
    38050
     600
312
      2
     540
     500
```

211

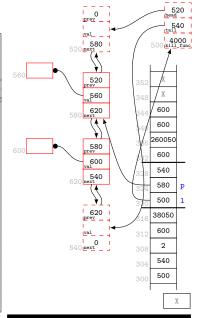
```
210
       * list kill() - Destroy a given list.
212
       * Ql: List to destroy.
213
214
       * Returns all dynamic memory used by the list and its el
215
       * kill func was registered at list creation, also calls
216
       * element to return any user-allocated memory occupied by
217
       * values.
218
219
       * Returns: Nothing.
220
221
      void list kill(list *1)
222
223
          // Use public functions to traverse the list.
224
225
          // Start with the first element (will be defined
226
          // even for an empty list).
227
          list_pos p = list_first(1);
228
229
          // Remove first element until list is empty.
230
          while (!list_is_empty(1)) {
              p = list_remove(1, p);
231
232
          }
233
234
          // Free border elements and the list head.
235
          free(1->head):
236
          free(1->tail);
237
          free(1);
238
```



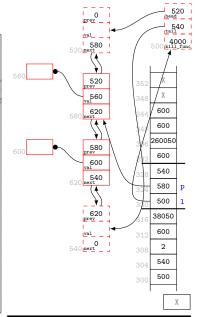
```
210
       * list kill() - Destroy a given list.
211
212
       * Ql: List to destroy.
213
214
       * Returns all dynamic memory used by the list and its el
215
       * kill func was registered at list creation, also calls
216
       * element to return any user-allocated memory occupied by
217
       * values.
218
219
       * Returns: Nothing.
220
       */
221
      void list kill(list *1)
222
      {
223
          // Use public functions to traverse the list.
224
225
          // Start with the first element (will be defined
226
          // even for an empty list).
227
          list_pos p = list_first(1);
228
229
          // Remove first element until list is empty.
230
          while (!list_is_empty(1)) {
              p = list_remove(1, p);
231
232
          }
233
234
          // Free border elements and the list head.
235
          free(1->head):
236
          free(1->tail);
237
          free(1);
238
```



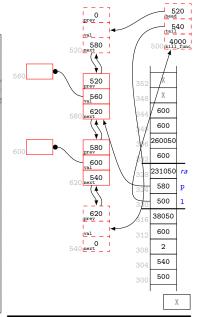
```
210
       * list kill() - Destroy a given list.
211
212
       * Ql: List to destroy.
213
214
       * Returns all dynamic memory used by the list and its el
215
       * kill func was registered at list creation, also calls
216
       * element to return any user-allocated memory occupied by
217
       * values.
218
219
       * Returns: Nothing.
220
       */
221
      void list kill(list *1)
222
      {
223
          // Use public functions to traverse the list.
224
225
          // Start with the first element (will be defined
226
          // even for an empty list).
227
          list_pos p = list_first(1);
228
229
          // Remove first element until list is empty.
230
          while (!list is empty(1)) {
231
             p = list_remove(1, p);
232
          }
233
234
          // Free border elements and the list head.
235
          free(1->head):
236
          free(1->tail);
237
          free(1);
238
```



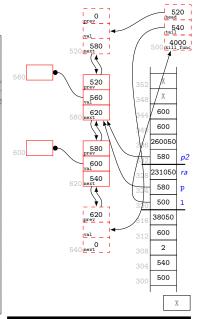
```
210
       * list kill() - Destroy a given list.
211
212
       * Ql: List to destroy.
213
214
       * Returns all dynamic memory used by the list and its el
215
       * kill func was registered at list creation, also calls
216
       * element to return any user-allocated memory occupied by
217
       * values.
218
219
       * Returns: Nothing.
220
       */
221
      void list kill(list *1)
222
      {
223
          // Use public functions to traverse the list.
224
225
          // Start with the first element (will be defined
226
          // even for an empty list).
227
          list_pos p = list_first(1);
228
229
          // Remove first element until list is empty.
230
          while (!list_is_empty(1)) {
231
              p = list_remove(1, p);
232
          }
233
234
          // Free border elements and the list head.
235
          free(1->head):
236
          free(1->tail);
237
          free(1);
238
```



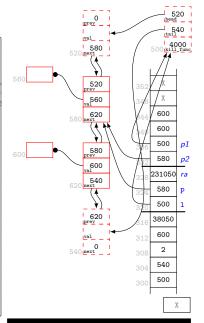
```
210
       * list kill() - Destroy a given list.
211
212
       * Ql: List to destroy.
213
214
       * Returns all dynamic memory used by the list and its el
215
       * kill func was registered at list creation, also calls
216
       * element to return any user-allocated memory occupied by
217
       * values.
218
219
       * Returns: Nothing.
220
       */
221
      void list kill(list *1)
222
      {
223
          // Use public functions to traverse the list.
224
225
          // Start with the first element (will be defined
226
          // even for an empty list).
227
          list_pos p = list_first(1);
228
229
          // Remove first element until list is empty.
230
          while (!list_is_empty(1)) {
231
              p = list_remove(1, p);
232
          }
233
234
          // Free border elements and the list head.
235
          free(1->head):
236
          free(1->tail);
237
          free(1);
238
```



```
210
       * list kill() - Destroy a given list.
211
212
       * Ql: List to destroy.
213
214
       * Returns all dynamic memory used by the list and its el
215
       * kill func was registered at list creation, also calls
216
       * element to return any user-allocated memory occupied by
217
       * values.
218
219
       * Returns: Nothing.
220
       */
221
      void list kill(list *1)
222
      {
223
          // Use public functions to traverse the list.
224
225
          // Start with the first element (will be defined
226
          // even for an empty list).
227
          list_pos p = list_first(1);
228
229
          // Remove first element until list is empty.
230
          while (!list_is_empty(1)) {
231
              p = list_remove(1, p);
232
          }
233
234
          // Free border elements and the list head.
235
          free(1->head):
236
          free(1->tail);
237
          free(1);
238
```



```
210
       * list kill() - Destroy a given list.
211
212
       * Ql: List to destroy.
213
214
       * Returns all dynamic memory used by the list and its el
215
       * kill func was registered at list creation, also calls
216
       * element to return any user-allocated memory occupied by
217
       * values.
218
219
       * Returns: Nothing.
220
       */
221
      void list kill(list *1)
222
      {
223
          // Use public functions to traverse the list.
224
225
          // Start with the first element (will be defined
226
          // even for an empty list).
227
          list_pos p = list_first(1);
228
229
          // Remove first element until list is empty.
230
          while (!list_is_empty(1)) {
231
              p = list_remove(1, p);
232
          }
233
234
          // Free border elements and the list head.
235
          free(1->head):
236
          free(1->tail);
237
          free(1);
238
```



180

181

182

183

184

185

186

187

188

189

190

191

192

193

194

195

196

197

198

199

200

201

202

203

204

205

206

207

208

```
* list remove() - Remove an element from a list.
 * Ql: List to manipulate.
 * Op: 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 *1, 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 (1->kill_func != NULL) {
        // Return any user-allocated memory for the value
       1->kill func(p->val):
    // Return the memory allocated to the cell itself.
    free(p):
    // Return the position of the next element.
    return np;
```

```
0
   prev
                            540
                          tail
                      4000
500 kill_func
     580
520 next
                         χ
     520
                         χ
     560
      620
                        600
                        600
                              np
                        500
      580
                        580
      600
                      231050
                  32
     540
620 next
                        580
                        500
     620
                      38050
    prev
                        600
      Ω
                         2
540 next
                        540
                        500
```

180

181

182

183

184

185

186

187

188

189

190

191

192

193

194

195

196

197

198

199

200

201

202

203

204

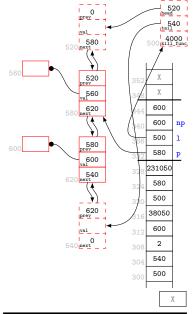
205

206

207

208

```
* list remove() - Remove an element from a list.
 * Ql: List to manipulate.
 * Op: 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 *1, 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 (1->kill_func != NULL) {
        // Return any user-allocated memory for the value
       1->kill func(p->val):
    // Return the memory allocated to the cell itself.
    free(p):
    // Return the position of the next element.
    return np;
```



180

181

182

183

184

185

186

187

188

189

190

191

192

193

194

195

196

197

198

199

200

201

202

203

204

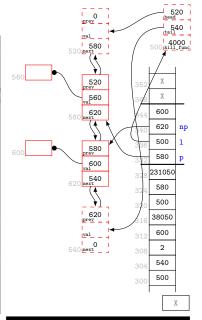
205

206

207

208

```
* list remove() - Remove an element from a list.
 * Ql: List to manipulate.
 * Op: 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 *1, 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 (1->kill_func != NULL) {
        // Return any user-allocated memory for the value
       1->kill func(p->val):
    // Return the memory allocated to the cell itself.
    free(p):
    // Return the position of the next element.
    return np;
```



180

181

182

183

184

185

186

187

188

189

190

191

192

193

194

195

196

197

198

199

200

201

202

203

204

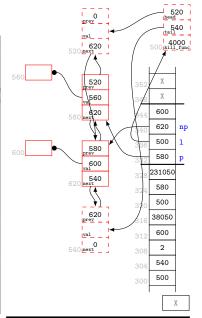
205

206

207

208

```
* list remove() - Remove an element from a list.
 * Ql: List to manipulate.
 * Op: 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 *1, 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 (1->kill_func != NULL) {
        // Return any user-allocated memory for the value
       1->kill func(p->val):
    // Return the memory allocated to the cell itself.
    free(p):
    // Return the position of the next element.
    return np;
```



180

181

182

183

184

185

186

187

188

189

190

191

192

193

194

195

196

197

198

199

200

201

202

203

204

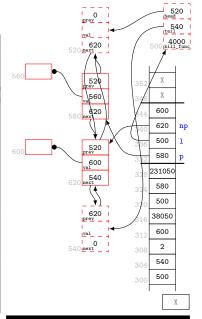
205

206

207

208

```
* list remove() - Remove an element from a list.
 * Ql: List to manipulate.
 * Op: 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 *1, 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 (1->kill_func != NULL) {
       // Return any user-allocated memory for the value
       1->kill func(p->val):
    // Return the memory allocated to the cell itself.
    free(p):
    // Return the position of the next element.
    return np;
```



180

181

182

183

184

185

186

187

188

189

190

191

192

193

194

195

196

197

198

199

200

201

202

203

204

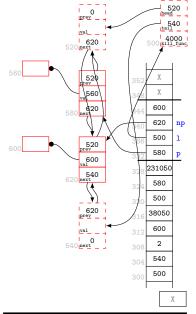
205

206

207

208

```
* list remove() - Remove an element from a list.
 * Ql: List to manipulate.
 * Op: 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 *1, 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 (1->kill_func != NULL) {
        // Return any user-allocated memory for the value
       1->kill func(p->val):
    // Return the memory allocated to the cell itself.
    free(p):
    // Return the position of the next element.
    return np;
```



180

181

182

183

184

185

186

187

188

189

190

191

192

193

194

195

196

197

198

199

200

201

202

203

204

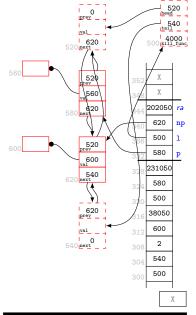
205

206

207

208

```
* list remove() - Remove an element from a list.
 * Ql: List to manipulate.
 * Op: 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 *1, 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 (1->kill_func != NULL) {
        // Return any user-allocated memory for the value
       1->kill func(p->val):
    // Return the memory allocated to the cell itself.
    free(p):
    // Return the position of the next element.
    return np;
```



180

181

182

183

184

185

186

187

188

189

190

191

192

193

194

195

196

197

198

199

200

 $\frac{201}{202}$

203

204

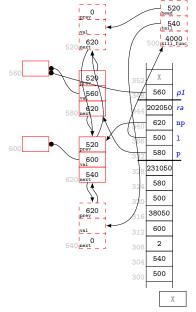
205

206

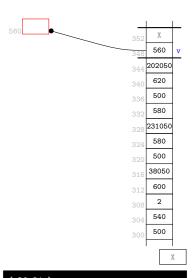
207

208

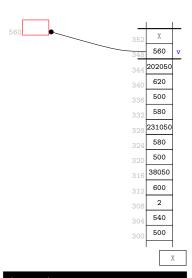
```
* list remove() - Remove an element from a list.
 * Ql: List to manipulate.
 * Op: Position in the list of the element to remove.
 * Removes the element at position p from the list. If a
 * was registered at list creation, calls it to deallocat
 * held by the element value.
 * Returns: The position after the removed element.
 */
list pos list remove(list *1, 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 (1->kill_func != NULL) {
        // Return any user-allocated memory for the value
       1->kill func(p->val):
    // Return the memory allocated to the cell itself.
    free(p):
    // Return the position of the next element.
    return np;
```



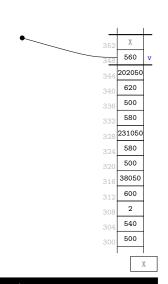
```
int *int_create(int v)
10
11
         int *p = malloc(sizeof(*p)); // allocate payload
12
         *p = v; // assign value
13
         return p;
14
15
     void int_kill(void *v)
16
         free(v); // deallocate payload
17
18
19
     int main(void)
20
     {
21
         // Create empty list, hand over responsibility to
22
         // deallocate payload using int kill
23
         list *l = list_empty(int_kill);
24
         list_pos p = list_first(1);
25
         for (int i = 0; i < 2; i++) {
26
             int *v = int_create(20 + i);
27
             p = list_insert(1, v, p);
28
             p = list next(1, p):
29
30
         list_print(1, print_int);
31
32
33
34
35
36
37
         // Clean up the list, including payload
38
         list_kill(1);
39
         return 0;
40
```



```
int *int_create(int v)
10
11
         int *p = malloc(sizeof(*p)); // allocate payload
12
         *p = v; // assign value
13
         return p;
14
15
     void int_kill(void *v)
16
17
         free(v); // deallocate payload
18
19
     int main(void)
20
     {
21
        // Create empty list, hand over responsibility to
22
         // deallocate payload using int kill
23
         list *l = list_empty(int_kill);
24
         list_pos p = list_first(1);
25
         for (int i = 0; i < 2; i++) {
26
             int *v = int_create(20 + i);
27
             p = list_insert(1, v, p);
28
             p = list next(1, p):
29
30
         list_print(1, print_int);
31
32
33
34
35
36
37
         // Clean up the list, including payload
38
         list_kill(1);
39
         return 0;
40
```



```
int *int_create(int v)
10
11
         int *p = malloc(sizeof(*p)); // allocate payload
12
         *p = v; // assign value
13
         return p;
14
15
     void int_kill(void *v)
16
17
         free(v); // deallocate payload
18
19
     int main(void)
20
     {
21
        // Create empty list, hand over responsibility to
22
         // deallocate payload using int kill
23
         list *l = list_empty(int_kill);
24
         list_pos p = list_first(1);
25
         for (int i = 0; i < 2; i++) {
26
             int *v = int_create(20 + i);
27
             p = list_insert(1, v, p);
28
             p = list next(1, p):
29
30
         list_print(1, print_int);
31
32
33
34
35
36
37
         // Clean up the list, including payload
38
         list_kill(1);
39
         return 0;
40
```



180

181

182

183

184

185

186

187

188

189

190

191

192

193

194

195

196

197

198

199

200

201

202

203

204

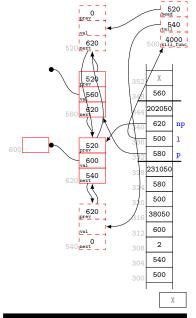
205

206

207

208

```
* list remove() - Remove an element from a list.
 * Ql: List to manipulate.
 * Op: 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 *1, 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 (1->kill_func != NULL) {
        // Return any user-allocated memory for the value
       1->kill func(p->val):
    // Return the memory allocated to the cell itself.
    free(p):
    // Return the position of the next element.
    return np;
```



180

181

182

183

184

185

186

187

188

189

190

191

192

193

194

195

196

197

198

199

200

201

202

203

204

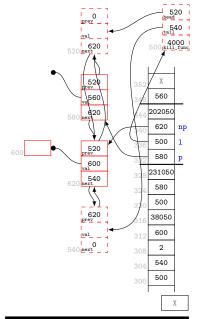
205

206

207

208

```
* list remove() - Remove an element from a list.
 * Ql: List to manipulate.
 * Op: 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 *1, 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 (1->kill_func != NULL) {
        // Return any user-allocated memory for the value
       1->kill func(p->val):
    // Return the memory allocated to the cell itself.
    free(p):
    // Return the position of the next element.
    return np;
```



180

181

182

183

184

185

186

187

188

189

190

191

192

193

194

195

196

197

198

199

200

201

202

203

204

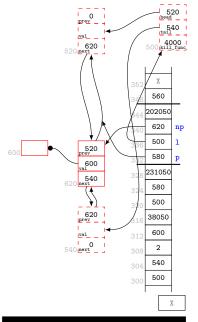
205

206

207

208

```
* list remove() - Remove an element from a list.
 * Ql: List to manipulate.
 * Op: 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 *1, 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 (1->kill_func != NULL) {
        // Return any user-allocated memory for the value
       1->kill func(p->val):
    // Return the memory allocated to the cell itself.
    free(p):
    // Return the position of the next element.
    return np;
```



180

181

182

183

184

185

186

187

188

189

190

191

192

193

194

195

196

197

198

199

200

201

202

203

204

205

206

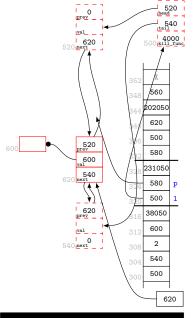
207

208

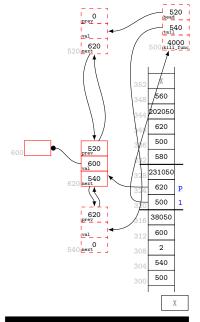
```
* list remove() - Remove an element from a list.
 * Ql: List to manipulate.
 * Op: Position in the list of the element to remove.
 * Removes the element at position p from the list. If a
 * was registered at list creation, calls it to deallocat
 * held by the element value.
 * Returns: The position after the removed element.
 */
list pos list remove(list *1, 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 (1->kill_func != NULL) {
        // Return any user-allocated memory for the value
       1->kill func(p->val):
    // Return the memory allocated to the cell itself.
    free(p):
    // Return the position of the next element.
    return np;
```

```
0
   prev
                           540
                         tail
                     4000
500 kill_func
     620
520 next
                        χ
                       560
                     202050
                       620
                             np
                       500
     520
                       580
     600
                     231050
                 32
     540
620 next
                       580
                       500
     620
                      38050
   prev
                       600
   val
      Ω
                        2
540 next
                       540
                       500
                           620
```

```
210
       * list kill() - Destroy a given list.
211
212
       * Ql: List to destroy.
213
214
       * Returns all dynamic memory used by the list and its el
215
       * kill func was registered at list creation, also calls
216
       * element to return any user-allocated memory occupied by
217
       * values.
218
219
       * Returns: Nothing.
220
       */
221
      void list kill(list *1)
222
      {
223
          // Use public functions to traverse the list.
224
225
          // Start with the first element (will be defined
226
          // even for an empty list).
227
          list_pos p = list_first(1);
228
229
          // Remove first element until list is empty.
230
          while (!list_is_empty(1)) {
231
              p = list_remove(1, p);
232
          }
233
234
          // Free border elements and the list head.
235
          free(1->head):
236
          free(1->tail);
237
          free(1);
238
```



```
210
       * list kill() - Destroy a given list.
211
212
       * Ql: List to destroy.
213
214
       * Returns all dynamic memory used by the list and its el
215
       * kill func was registered at list creation, also calls
216
       * element to return any user-allocated memory occupied by
217
       * values.
218
219
       * Returns: Nothing.
220
       */
221
      void list kill(list *1)
222
      {
223
          // Use public functions to traverse the list.
224
225
          // Start with the first element (will be defined
226
          // even for an empty list).
227
          list_pos p = list_first(1);
228
229
          // Remove first element until list is empty.
230
          while (!list is empty(1)) {
231
             p = list_remove(1, p);
232
          }
233
234
          // Free border elements and the list head.
235
          free(1->head):
236
          free(1->tail);
237
          free(1);
238
```



210

211 212

218

221

223

224

225

226

227

228

229

230

231

232

233

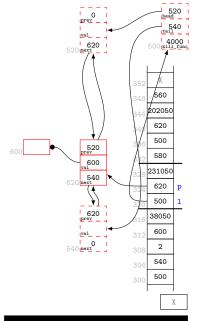
235

236

237

238

```
* list kill() - Destroy a given list.
       * Ql: List to destroy.
213
214
       * Returns all dynamic memory used by the list and its el
215
       * kill func was registered at list creation, also calls
216
       * element to return any user-allocated memory occupied by
217
       * values.
219
       * Returns: Nothing.
220
       */
      void list kill(list *1)
222
      {
          // 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(1);
          // Remove first element until list is empty.
          while (!list_is_empty(1)) {
              p = list_remove(1, p);
          }
234
          // Free border elements and the list head.
          free(1->head):
          free(1->tail);
          free(1);
```



210

211 212

217

218

221

223

224

225

226

227

228

229

230

231

232

233

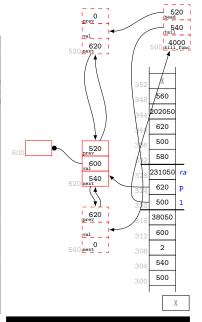
235

236

237

238

```
* list kill() - Destroy a given list.
       * Ql: List to destroy.
213
214
       * Returns all dynamic memory used by the list and its el
215
       * kill func was registered at list creation, also calls
216
       * element to return any user-allocated memory occupied by
       * values.
219
       * Returns: Nothing.
220
       */
      void list kill(list *1)
222
      {
          // 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(1);
          // Remove first element until list is empty.
          while (!list_is_empty(1)) {
              p = list_remove(1, p);
          }
234
          // Free border elements and the list head.
          free(1->head):
          free(1->tail);
          free(1);
```



210

211 212

217

218

221

223

224

225

226

227

228

229

230

231

232

233

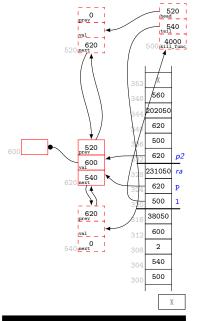
235

236

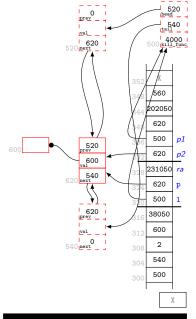
237

238

```
* list kill() - Destroy a given list.
       * Ql: List to destroy.
213
214
       * Returns all dynamic memory used by the list and its el
215
       * kill func was registered at list creation, also calls
216
       * element to return any user-allocated memory occupied by
       * values.
219
       * Returns: Nothing.
220
       */
      void list kill(list *1)
222
      {
          // 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(1);
          // Remove first element until list is empty.
          while (!list_is_empty(1)) {
              p = list_remove(1, p);
          }
234
          // Free border elements and the list head.
          free(1->head):
          free(1->tail);
          free(1);
```



```
210
       * list kill() - Destroy a given list.
211
212
       * Ql: List to destroy.
213
214
       * Returns all dynamic memory used by the list and its el
215
       * kill func was registered at list creation, also calls
216
       * element to return any user-allocated memory occupied by
217
       * values.
218
219
       * Returns: Nothing.
220
       */
221
      void list kill(list *1)
222
      {
223
          // Use public functions to traverse the list.
224
225
          // Start with the first element (will be defined
226
          // even for an empty list).
227
          list_pos p = list_first(1);
228
229
          // Remove first element until list is empty.
230
          while (!list_is_empty(1)) {
231
              p = list_remove(1, p);
232
          }
233
234
          // Free border elements and the list head.
235
          free(1->head):
236
          free(1->tail);
237
          free(1);
238
```



180

181

182

183

184

185

186

187

188

189

190

191

192

193

194

195

196

197

198

199

200

201

202

203

204

205

206

207

208

```
* list remove() - Remove an element from a list.
 * Ql: List to manipulate.
 * Op: 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 *1, 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 (1->kill_func != NULL) {
        // Return any user-allocated memory for the value
       1->kill func(p->val):
    // Return the memory allocated to the cell itself.
    free(p):
    // Return the position of the next element.
    return np;
```

```
0
   prev
                            540
                      4000
500 kill_func
     620
520 next
                         χ
                        560
                      202050
                        620
                              np
                        500
      520
                        620
      600
                      231050
                  32
     540
620 next
                        620
                        500
     620
                       38050
    prev
                        600
      Ω
                         2
540 next
                        540
                        500
```

180

181

182

183

184

185

186

187

188

189

190

191

192

193

194

195

196

197

198

199

200

201

202

203

204

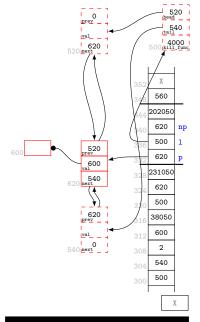
205

206

207

208

```
* list remove() - Remove an element from a list.
 * Ql: List to manipulate.
 * Op: 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 *1, 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 (1->kill_func != NULL) {
        // Return any user-allocated memory for the value
       1->kill func(p->val):
    // Return the memory allocated to the cell itself.
    free(p):
    // Return the position of the next element.
    return np;
```



180

181

182

183

184

185

186

187

188

189

190

191

192

193

194

195

196

197

198

199

200

201

202

203

204

205

206

207

208

```
* list remove() - Remove an element from a list.
 * Ql: List to manipulate.
 * Op: 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 *1, 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 (1->kill_func != NULL) {
        // Return any user-allocated memory for the value
       1->kill func(p->val):
    // Return the memory allocated to the cell itself.
    free(p):
    // Return the position of the next element.
    return np;
```

```
520
      0
   prev
                           540
                         tail
                      4000
500 kill_func
     620
520 next
                        χ
                       560
                      202050
                       540
                             np
                       500
     520
                       620
     600
                     231050
                 32
     540
620 next
                       620
                       500
     620
                      38050
   prev
                       600
      Ω
                        2
540 next
                       540
                       500
```

180

181

182

183

184

185

186

187

188

189

190

191

192

193

194

195

196

197

198

199

200

201

202

203

204

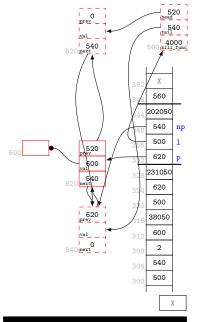
205

206

207

208

```
* list remove() - Remove an element from a list.
 * Ql: List to manipulate.
 * Op: 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 *1, 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 (1->kill_func != NULL) {
        // Return any user-allocated memory for the value
       1->kill func(p->val):
    // Return the memory allocated to the cell itself.
    free(p):
    // Return the position of the next element.
    return np;
```



180

181

182

191

193

195

196

197

198

199

200

201

202

203

204

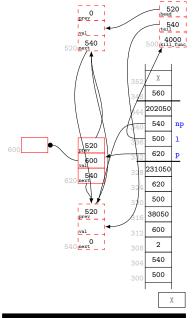
205

206

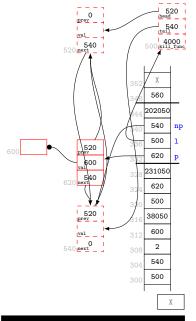
207

208

```
* list remove() - Remove an element from a list.
       * Ql: List to manipulate.
183
       * Op: Position in the list of the element to remove.
184
185
       * Removes the element at position p from the list. If a
186
       * was registered at list creation, calls it to deallocat
187
       * held by the element value.
188
189
       * Returns: The position after the removed element.
190
       */
      list_pos list_remove(list *1, const list_pos p)
192
          // Remember return position.
194
          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 (1->kill_func != NULL) {
             // Return any user-allocated memory for the value
             1->kill func(p->val):
          // Return the memory allocated to the cell itself.
          free(p):
          // Return the position of the next element.
          return np;
```



```
180
181
       * list remove() - Remove an element from a list.
182
       * Ql: List to manipulate.
183
       * Op: Position in the list of the element to remove.
184
185
       * Removes the element at position p from the list. If a
186
       * was registered at list creation, calls it to deallocat
187
       * held by the element value.
188
189
       * Returns: The position after the removed element.
190
       */
191
      list pos list remove(list *1, const list pos p)
192
193
          // Remember return position.
194
          list_pos np = p->next;
195
          // Link past this element.
196
          p->prev->next = p->next;
197
          p->next->prev = p->prev;
198
199
          // Call kill func if registered.
200
          if (1->kill_func != NULL) {
              // Return any user-allocated memory for the value
201
202
             1->kill func(p->val):
203
204
          // Return the memory allocated to the cell itself.
205
          free(p):
206
          // Return the position of the next element.
207
          return np;
208
```



180

181

182

183

184

185

186

187

188

189

190

191

192

193

194

195

196

197

198

199

200

201

202

203

204

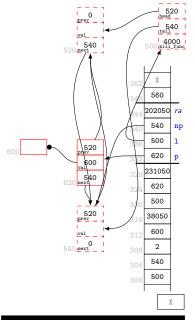
205

206

207

208

```
* list remove() - Remove an element from a list.
 * Ql: List to manipulate.
 * Op: 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 *1, 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 (1->kill_func != NULL) {
        // Return any user-allocated memory for the value
       1->kill func(p->val):
    // Return the memory allocated to the cell itself.
    free(p):
    // Return the position of the next element.
    return np;
```



180

181

182

183

184

185

186

187

188

189

190

191

192

193

194

195

196

197

198

199

200

201

202

203

204

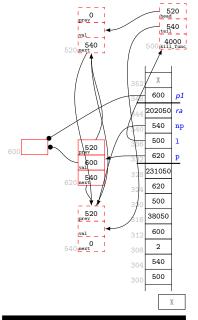
205

206

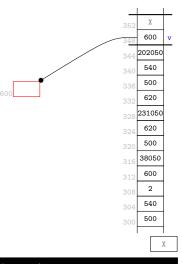
207

208

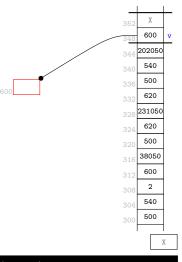
```
* list remove() - Remove an element from a list.
 * Ql: List to manipulate.
 * On: Position in the list of the element to remove.
 * Removes the element at position p from the list. If a
 * was registered at list creation, calls it to deallocat
 * held by the element value.
 * Returns: The position after the removed element.
 */
list pos list remove(list *1, 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 (1->kill_func != NULL) {
        // Return any user-allocated memory for the value
       1->kill func(p->val):
    // Return the memory allocated to the cell itself.
    free(p):
    // Return the position of the next element.
    return np;
```



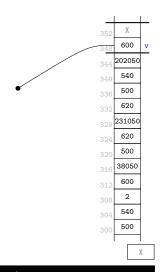
```
int *int_create(int v)
10
11
         int *p = malloc(sizeof(*p)); // allocate payload
12
         *p = v; // assign value
13
         return p;
14
15
     void int_kill(void *v)
16
         free(v); // deallocate payload
17
18
19
     int main(void)
20
     {
21
         // Create empty list, hand over responsibility to
22
         // deallocate payload using int kill
23
         list *l = list_empty(int_kill);
24
         list_pos p = list_first(1);
25
         for (int i = 0; i < 2; i++) {
26
             int *v = int_create(20 + i);
27
             p = list_insert(1, v, p);
28
             p = list next(1, p):
29
30
         list_print(1, print_int);
31
32
33
34
35
36
37
         // Clean up the list, including payload
38
         list_kill(1);
39
         return 0;
40
```



```
int *int_create(int v)
10
11
         int *p = malloc(sizeof(*p)); // allocate payload
12
         *p = v; // assign value
13
         return p;
14
15
     void int_kill(void *v)
16
17
         free(v); // deallocate payload
18
19
     int main(void)
20
     {
21
        // Create empty list, hand over responsibility to
        // deallocate payload using int kill
22
23
         list *l = list_empty(int_kill);
24
         list_pos p = list_first(1);
25
         for (int i = 0; i < 2; i++) {
26
             int *v = int_create(20 + i);
27
             p = list_insert(1, v, p);
28
             p = list next(1, p):
29
30
         list_print(1, print_int);
31
32
33
34
35
36
37
         // Clean up the list, including payload
38
         list_kill(1);
39
         return 0;
40
```



```
int *int_create(int v)
10
11
         int *p = malloc(sizeof(*p)); // allocate payload
12
         *p = v; // assign value
13
         return p;
14
15
     void int_kill(void *v)
16
17
         free(v); // deallocate payload
18
19
     int main(void)
20
     {
21
         // Create empty list, hand over responsibility to
22
         // deallocate payload using int kill
23
         list *l = list_empty(int_kill);
24
         list_pos p = list_first(1);
25
         for (int i = 0; i < 2; i++) {
26
             int *v = int_create(20 + i);
27
             p = list_insert(1, v, p);
28
             p = list next(1, p):
29
30
         list_print(1, print_int);
31
32
33
34
35
36
37
         // Clean up the list, including payload
38
         list_kill(1);
39
         return 0;
40
```



180

181

182

183

184

185

186

187

188

189

190

191

192

193

194

195

196

197

198

199

200

201

202

203

204

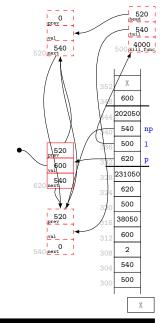
205

206

207

208

```
* list remove() - Remove an element from a list.
 * Ql: List to manipulate.
 * Op: 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 *1, 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 (1->kill_func != NULL) {
        // Return any user-allocated memory for the value
       1->kill func(p->val):
    // Return the memory allocated to the cell itself.
    free(p):
    // Return the position of the next element.
    return np;
```



180

181

182

183

184

185

186

187

188

189

190

191

192

193

194

195

196

197

198

199

200

201

202

203

204

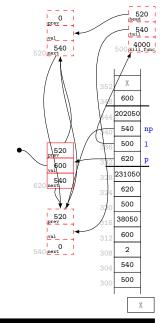
205

206

207

208

```
* list remove() - Remove an element from a list.
 * Ql: List to manipulate.
 * Op: 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 *1, 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 (1->kill_func != NULL) {
        // Return any user-allocated memory for the value
       1->kill func(p->val):
    // Return the memory allocated to the cell itself.
    free(p):
    // Return the position of the next element.
    return np;
```



180

181

191

193

195

196

197

198

200

201

202

203

204

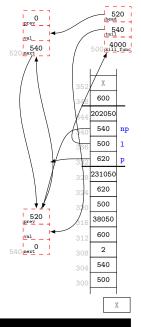
205

206

207

208

```
* list remove() - Remove an element from a list.
182
       * Ql: List to manipulate.
183
       * Op: Position in the list of the element to remove.
184
185
       * Removes the element at position p from the list. If a
186
       * was registered at list creation, calls it to deallocat
187
       * held by the element value.
188
189
       * Returns: The position after the removed element.
190
       */
      list pos list remove(list *1, const list pos p)
192
          // Remember return position.
194
          list_pos np = p->next;
          // Link past this element.
          p->prev->next = p->next;
          p->next->prev = p->prev;
199
          // Call kill func if registered.
          if (1->kill_func != NULL) {
              // Return any user-allocated memory for the value
             1->kill func(p->val):
          7
          // Return the memory allocated to the cell itself.
          free(p):
          // Return the position of the next element.
          return np;
```



180

181

182

183

184

185

186

187

188

189

190

191

192

193

194

195

196

197

198

199

200

201

202

203

204

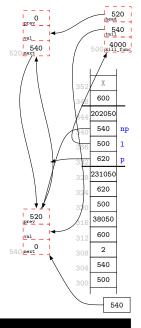
205

206

207

208

```
* list remove() - Remove an element from a list.
 * Ql: List to manipulate.
 * Op: 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 *1, 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 (1->kill_func != NULL) {
        // Return any user-allocated memory for the value
       1->kill func(p->val):
    // Return the memory allocated to the cell itself.
    free(p):
    // Return the position of the next element.
    return np;
```



210

211 212

221

223

224 225

226

227

228

229

230

231

232

233

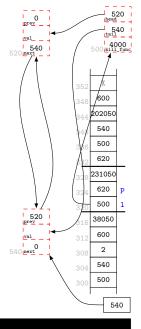
235

236

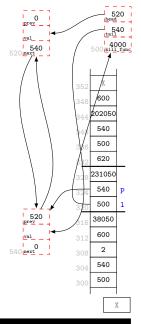
237

238

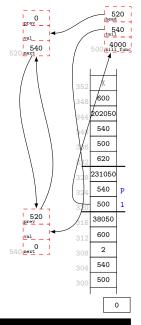
```
* list kill() - Destroy a given list.
       * Ql: List to destroy.
213
214
       * Returns all dynamic memory used by the list and its el
215
       * kill func was registered at list creation, also calls
216
       * element to return any user-allocated memory occupied by
217
       * values.
218
219
       * Returns: Nothing.
220
       */
      void list kill(list *1)
222
      {
          // 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(1);
          // Remove first element until list is empty.
          while (!list_is_empty(1)) {
              p = list_remove(1, p);
          }
234
          // Free border elements and the list head.
          free(1->head):
          free(1->tail);
          free(1);
```



```
210
       * list kill() - Destroy a given list.
211
212
       * Ql: List to destroy.
213
214
       * Returns all dynamic memory used by the list and its el
215
       * kill func was registered at list creation, also calls
216
       * element to return any user-allocated memory occupied by
217
       * values.
218
219
       * Returns: Nothing.
220
       */
221
      void list kill(list *1)
222
      {
223
          // Use public functions to traverse the list.
224
225
          // Start with the first element (will be defined
226
          // even for an empty list).
227
          list_pos p = list_first(1);
228
229
          // Remove first element until list is empty.
230
          while (!list is empty(1)) {
231
             p = list_remove(1, p);
232
          }
233
234
          // Free border elements and the list head.
235
          free(1->head):
236
          free(1->tail);
237
          free(1);
238
```



```
210
       * list kill() - Destroy a given list.
211
212
       * Ql: List to destroy.
213
214
       * Returns all dynamic memory used by the list and its el
215
       * kill func was registered at list creation, also calls
216
       * element to return any user-allocated memory occupied by
217
       * values.
218
219
       * Returns: Nothing.
220
       */
221
      void list kill(list *1)
222
      {
223
          // Use public functions to traverse the list.
224
225
          // Start with the first element (will be defined
226
          // even for an empty list).
227
          list_pos p = list_first(1);
228
229
          // Remove first element until list is empty.
230
          while (!list_is_empty(1)) {
231
              p = list_remove(1, p);
232
          }
233
234
          // Free border elements and the list head.
235
          free(1->head):
236
          free(1->tail);
237
          free(1);
238
```



210

211 212

221

223

224 225

226

227

228

231 232

233

234

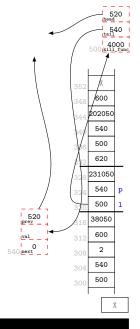
235

236

237

238

```
* list kill() - Destroy a given list.
       * Ql: List to destroy.
213
214
       * Returns all dynamic memory used by the list and its el
215
       * kill func was registered at list creation, also calls
216
       * element to return any user-allocated memory occupied by
217
       * values.
218
219
       * Returns: Nothing.
220
       */
      void list kill(list *1)
222
      {
          // 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(1);
229
          // Remove first element until list is empty.
230
          while (!list_is_empty(1)) {
              p = list_remove(1, p);
          }
          // Free border elements and the list head.
          free(1->head):
          free(1->tail);
          free(1);
```



210

211 212

213

215

216

217

218

219

220

221

223

224 225

226

227

228

229

230

231

232

233

234

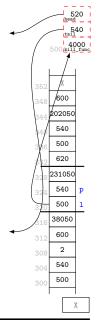
235

236

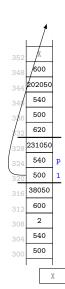
237

238

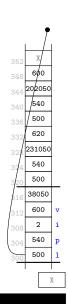
```
* list kill() - Destroy a given list.
       * Ql: List to destroy.
214
       * Returns all dynamic memory used by the list and its el
       * kill func was registered at list creation, also calls
       * element to return any user-allocated memory occupied by
       * values.
       * Returns: Nothing.
       */
      void list kill(list *1)
222
      {
          // 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(1);
          // Remove first element until list is empty.
          while (!list_is_empty(1)) {
              p = list_remove(1, p);
          }
          // Free border elements and the list head.
          free(1->head):
          free(1->tail);
          free(1);
```



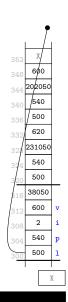
```
210
       * list kill() - Destroy a given list.
211
212
       * Ql: List to destroy.
213
214
       * Returns all dynamic memory used by the list and its el
215
       * kill_func was registered at list creation, also calls
216
       * element to return any user-allocated memory occupied by
217
       * values.
218
219
       * Returns: Nothing.
220
       */
221
      void list kill(list *1)
222
      {
223
          // Use public functions to traverse the list.
224
225
          // Start with the first element (will be defined
226
          // even for an empty list).
227
          list_pos p = list_first(1);
228
229
          // Remove first element until list is empty.
230
          while (!list_is_empty(1)) {
             p = list_remove(1, p);
231
232
          }
233
234
          // Free border elements and the list head.
235
          free(1->head):
236
          free(1->tail);
237
          free(1);
238
```



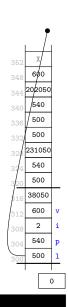
```
int *int_create(int v)
10
         int *p = malloc(sizeof(*p)); // allocate payload
11
12
        *p = v; // assign value
13
         return p;
14
15
     void int_kill(void *v)
16
17
         free(v); // deallocate payload
18
19
     int main(void)
20
     {
21
         // Create empty list, hand over responsibility to
22
         // deallocate payload using int kill
23
         list *l = list_empty(int_kill);
24
         list_pos p = list_first(1);
25
         for (int i = 0; i < 2; i++) {
26
             int *v = int_create(20 + i);
27
             p = list_insert(1, v, p);
28
             p = list next(1, p):
29
30
         list_print(1, print_int);
31
32
33
34
35
36
37
         // Clean up the list, including payload
38
         list_kill(1);
39
         return 0;
40
```



```
int *int_create(int v)
10
         int *p = malloc(sizeof(*p)); // allocate payload
11
12
         *p = v; // assign value
13
         return p;
14
15
     void int_kill(void *v)
16
17
         free(v); // deallocate payload
18
19
     int main(void)
20
     {
21
         // Create empty list, hand over responsibility to
22
         // deallocate payload using int kill
23
         list *l = list_empty(int_kill);
24
         list_pos p = list_first(1);
25
         for (int i = 0; i < 2; i++) {
26
             int *v = int_create(20 + i);
27
             p = list_insert(1, v, p);
28
             p = list next(1, p):
29
30
         list_print(1, print_int);
31
32
33
34
35
36
37
         // Clean up the list, including payload
38
         list_kill(1);
39
         return 0;
40
```



```
int *int_create(int v)
10
         int *p = malloc(sizeof(*p)); // allocate payload
11
12
         *p = v; // assign value
13
         return p;
14
15
     void int_kill(void *v)
16
17
         free(v); // deallocate payload
18
19
     int main(void)
20
     {
21
         // Create empty list, hand over responsibility to
22
         // deallocate payload using int kill
23
         list *l = list_empty(int_kill);
24
         list_pos p = list_first(1);
25
         for (int i = 0; i < 2; i++) {
26
             int *v = int_create(20 + i);
27
             p = list_insert(1, v, p);
28
             p = list next(1, p):
29
30
         list_print(1, print_int);
31
32
33
34
35
36
37
         // Clean up the list, including payload
38
         list_kill(1);
39
         return 0;
40
```



```
int *int_create(int v)
10
         int *p = malloc(sizeof(*p)); // allocate payload
11
12
         *p = v; // assign value
13
         return p;
14
15
     void int_kill(void *v)
16
17
         free(v); // deallocate payload
18
19
     int main(void)
20
     {
21
        // Create empty list, hand over responsibility to
22
         // deallocate payload using int kill
23
         list *l = list_empty(int_kill);
24
         list_pos p = list_first(1);
25
         for (int i = 0; i < 2; i++) {
26
             int *v = int_create(20 + i);
27
             p = list_insert(1, v, p);
28
             p = list next(1, p):
29
30
         list_print(1, print_int);
31
32
33
34
35
36
37
         // Clean up the list, including payload
38
         list_kill(1);
39
         return 0;
40
```

```
X
 600
202050
  540
  500
 500
231050
 540
 500
38050
 600
   2
  540
 500
```

Ansvarsförvirring

Ansvarsförvirring

- Ansvaret f\u00f6r att l\u00e4mna 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:
 - Ansvaret f\u00f6r att \u00e5terl\u00e4mna minnet beh\u00e5lls av den del av programmet som reserverade minnet
 - Vi kan kalla det för att listan lånat minnet
 - Aterlämningen sker i kod som ligger utanför listan
 - 2. Listan övertar ansvaret för att återlämna minnet
 - ► Vi kan kalla det för att listan äger minnet
 - Aterlämningen 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 *1 = 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 *1 = 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 datat (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

19

20

21

22

23

24

25

26

27

28

29

30

31

32

33

34

35

36

37

38

39

40

```
kill function
       code/list-with-freehandler.c
int main(void)
                                               19
                                               20
    // Create empty list, hand over responsibil 2th to
   // deallocate payload using int kill
                                               22
   list *1 = list empty(int kill):
                                               23
   list pos p = list first(1):
                                               24
   for (int i = 0; i < 2; i++) {
                                               25
        int *v = int_create(20 + i);
                                               26
       p = list_insert(1, v, p);
                                               27
       p = list_next(1, p);
                                               28
                                               29
   list_print(1, print_int);
                                               30
                                               31
                                               32
                                               33
                                               34
                                               35
                                               36
    // Clean up the list, including payload
                                               37
   list_kill(1);
                                               38
                                               39
   return 0:
                                               40
```

Lista 3b använder ingen

kill function

```
code/list-no-freehandler.c
int main(void)
    // Create empty list, keep responsibility to
    // deallocate payload
    list *1 = list emptv(NULL):
    list_pos p = list_first(1);
   for (int i = 0; i < 2; i++) {
        int *v = int_create(20 + i);
        p = list_insert(1, v, p);
        p = list_next(1, p);
    list_print(1, print_int);
    // Clean up the payloads
    p = list_first(1);
    while (!list_pos_is_equal(1, p, list_end(1)))
        int *v = list inspect(l, p):
        int kill(v); // free payload
        p = list next(1, p);
    list_kill(1); // Clean up the list
    return 0:
```

Motsvarar grovt list mwe2.c

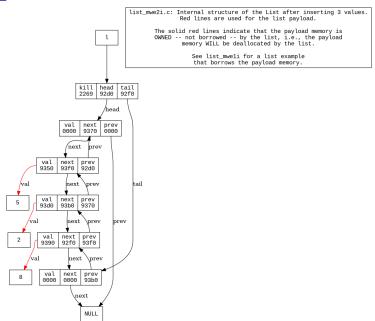
list_mwe1.c

Motsvarar grovt

Blank

list mwe2i.c

Niclas Börlin — 5DV149, DoA-C



list mwe1i.c

Niclas Börlin — 5DV149, DoA-C

