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

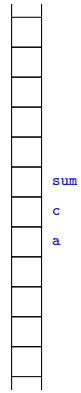
```
1  #include <stdio.h>
2  void print(int v)
3  {
4      printf("v = %d\n",v);
5  }
6  int add(int c, int d)
7  {
8      int a;
9      a = c + d;
10     return a;
11 }
12 int main(void)
13 {
14     int a = 2;
15     int c = 3;
16     int sum;
17     sum = add(a, c);
18     print(sum);
19     sum =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


Lokala variabler och stacken (1)

- Lokala variabler lagras i en minnesarea som kallas stacken

code/vars-on-stack.c

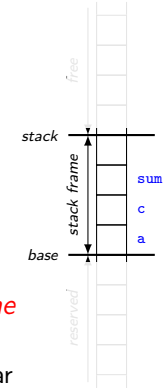
```
12 int main(void)
13 {
14     int a = 2;
15     int c = 3;
16     int sum;
```



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

Lokala variabler och stacken (2)

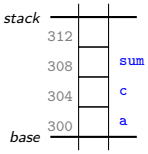
- Internt används två pekare (*base* och *stack*) för att hålla reda på funktionens del av stacken
  - Området mellan pekarna kallas för *stack frame* (aktiveringspost)
  - Området under kan betraktas som upptaget
  - Området ovanför kan betraktas som ledigt
- När funktionen körs är endast minnet inom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



Lokala variabler, minne (1)

code/vars-on-stack.c

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



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

Lokala variabler, minne (2)

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

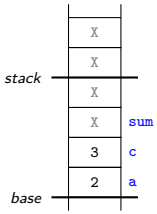
  men inte
  - variabelns adress
  - variabelns storlek

Funktionsanrop (1)

- Stacken används också för parameteröverföring vid funktionsanrop

code/vars-on-stack.c

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



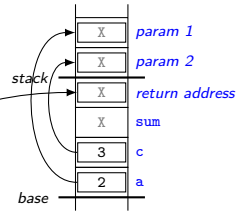
- Koden på rad 17 översätts av kompilatorn till ungefär följande operationer
- Den översatta maskinkoden kommer att hamna någonstans i minnet
  - Jag illustrerar med fejkade adresser som börjar på radnumret
    - Varje instruktion antas ta 10 bytes
      1) Beräkna och lagra en returadress på stacken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

Funktionsanrop (2)

- Stacken används också för parameteröverföring vid funktionsanrop

code/vars-on-stack.c

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



- I mer detalj så översätts instruktionerna

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

till

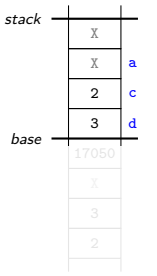
```
17010   store 17050 at (stack-4)    // ret addr
17020   store (base+4) at (stack+0) // c -> p2
17030   store (base+0) at (stack+4) // a -> p1
17040   call add
17050
```


Funktionsanrop (3)

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

code/vars-on-stack.c

```
6  int add(int c, int d)
7  {
8      int a;
9      a = c + d;
10     return a;
11 }
```



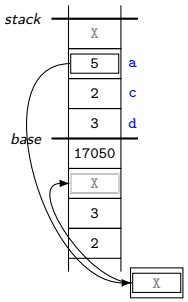
- ...och reserverar en egen *stack frame* i den fria delen av stacken
- Den anropande funktionens variabler blir osynliga och skyddas
- Notera att parametrarna `c` och `d` fungerar som lokala variabler
  - Parametrarna är initierade (har giltiga värden) när funktionen startar
  - Övriga lokala variabler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

Återhopp

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

code/vars-on-stack.c

```
6  int add(int c, int d)
7  {
8      int a;
9      a = c + d;
10     return a;
11 }
12 int main(void)
13 {
14     int a = 2;
15     int c = 3;
16     int sum;
17     sum = add(a, c);
```



- ...exekveringen fortsätter vid återhoppsadressen...

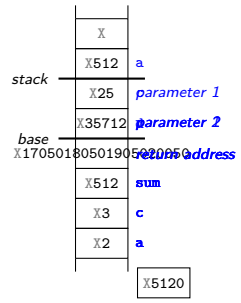
```
17050   store register at (base+8) // sum
```

- ...och sedan vidare på nästa rad...

Exemplet steg för steg

code/vars-on-stack.c

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



Rekursion (1)

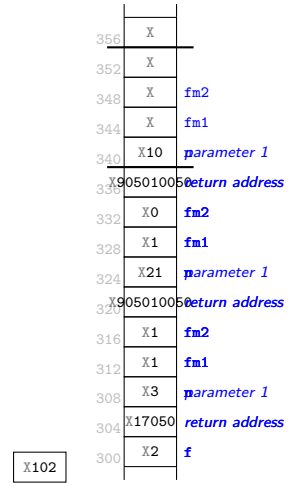
- Här är ett rekursivt exempel som beräknar Fibonacci-talen som definieras

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

```
1  #include <stdio.h>
2
3  int fib(int n)
4  {
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 }
```


Rekursion (2)

```
1  #include <stdio.h>
2
3  int fib(int n)
4  {
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 }
```




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

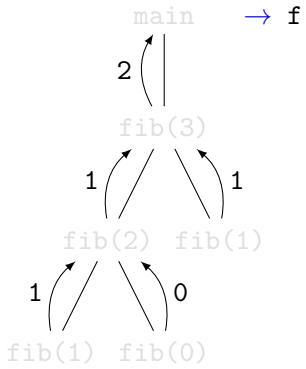
Rekursion (3)

code/fib.c

```c
#include <stdio.h>

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

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



code/fib.c

```c
#include <stdio.h>

int fib(int n)
{
    int fm1, fm2;
    if (n < 2) {
        return n;
    } else {
```


Blank

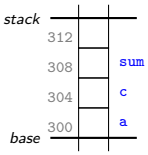
Adresser och pekare

Adress-operatorn

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

code/vars-on-stack.c

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



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

Pekare (1)

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

Pekare (2)

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

Pekare (3)

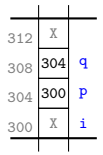
- Inget hindrar att vi har en pekare till en pekare
  - `int **q;`
  - Här är variabeln `q` av typen "pekare till pekare till `int`" eller "dubbelpekare till `int`" (*"`int` double pointer"*)
- Notera att om flera variabler deklareras i samma sats så är stjärnan kopplad till variabeln, inte till typen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

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

Pekare och adress-operatorn

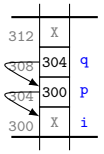
- En pekare till typen X kan tilldelas adressen för en variabel av typen X

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



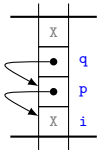
- Ofta illustrerar man pekare med hjälp av en pil

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



- Oftast utelämnar man adresserna helt

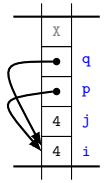
```
int i, *p, **q;
p = &i;
q = &p;
```


```

Dereferering

- Studera koden

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



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

Pekare som parametrar (1)

- Vad kommer att skrivas ut av den här koden?

code/add-one1.c

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

- ...och den här?

code/add-one2.c

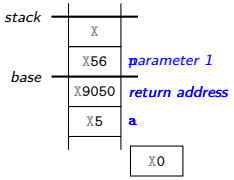
```c
#include <stdio.h>
void add_one(int *n)
{
    *n = *n + 1;
}
int main(void)
{
    int a = 5;
    add_one(&a);
    printf("a = %d\n", a);
    return 0;
}
```


Pekare som parametrar (2)

- Vi testkör!

code/add-one1.c

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

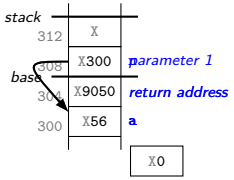


Pekare som parametrar (3)

- Vi testkör det andra examplet!

code/add-one2.c

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



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:

```
1  void swap(int *v1, int *v2)
2  {
3      int d = *v1;
4      *v1 = *v2;
5      *v2 = d;
6  }
```

  - 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

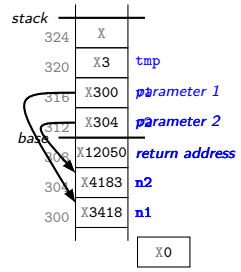
```
if (scanf("%d,%d", &a, &b) == 2) {
    // We have good values in a and b
} else {
    // Do error handling
}
```

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

Ett till exempel

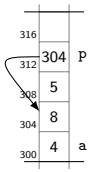
code/swap.c

```
1  #include <stdio.h>
2  void swap(int *v1, int *v2)
3  {
4      int tmp;
5      tmp = *v1;
6      *v1 = *v2;
7      *v2 = tmp;
8  }
9  int main(void)
10 {
11     int n1 = 3, n2 = 418;
12     swap(&n1, &n2);
13     printf("n1 = %4.2f, n2 = %4.2f\n", n1, n2);
14     return 0;
15 }
```



Pekare och fält

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



Pekare och poster (1)

- Vi kan också ha pekare till poster (`struct`)

code/struct_on_stack.c

```
2  typedef struct node {
3      int val;
4      struct node *next;
5  } 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

```
2  typedef struct node {
3      int val;
4      struct node *next;
5  } node;
```

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

Länkade strukturer

Länkade listor

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

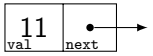
Lista 1: Typad payload, noder som lokala variabler (1)

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

code/struct-on-stack.c

```
2  typedef struct node {
3      int val; // typed payload of fixed size inside struct
4      struct node *next;
5  } 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

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

code/struct-on-stack.c

```
6  void print_list(const node *p)
7  {
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 }
```


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

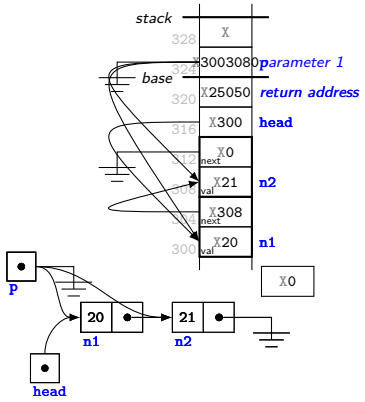
code/struct-on-stack.c

```
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
```

- Allokering och avallokering kommer att ske automatiskt

Lista 1: Visualisering

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




```
( 20 21 )
```


Begränsningar

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

Dynamisk minneshantering i C

Dynamisk minneshantering i C (1)

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

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

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

Dynamisk minneshantering i C (2)

- Reservera minne till en heltalsvektor med 10 element:

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

  - Operatorn `sizeof` returnerar storleken på argumentet i bytes

Dynamisk minneshantering i C (3)

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

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

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

Minnesballonger

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

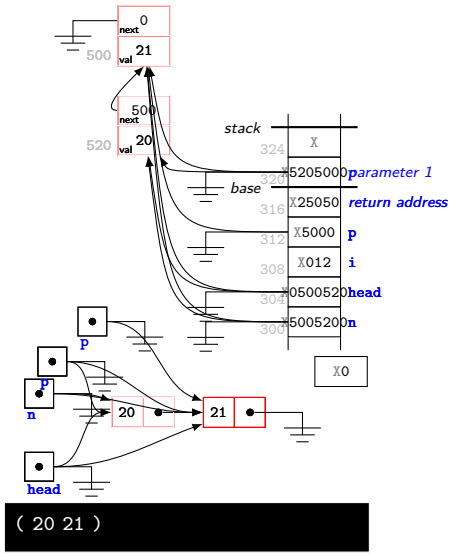
Lista 2: Typad payload, dynamiska noder

- I lista 2 allokeras och avallokeras noderna dynamiskt

code/dynamic-struct.c
```
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 }
```

Lista 2: Typad payload, dynamiska noder

```
3  typedef struct node {
4      int val; // typed payload of fixed size inside struct
5      struct node *next;
6  } node;
7  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 }
```



Generiska datatyper i C

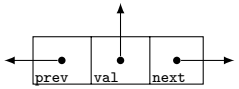
Generiska datatyper

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

code/list.c

```
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

code/list.h

```
32 // List type.
33 typedef struct list list;
34
35 // List position type.
36 typedef struct cell *list_pos;
```


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

```
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(l);
25     for (int i = 0; i < 2; i++) {
26         int *v = int_create(20 + i);
27         p = list_insert(l, v, p);
28         p = list_next(l, p);
29     }
30     list_print(l, print_int);
31
32
33
34
35
36
37     // Clean up the list, including payload
38     list_kill(l);
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

code/list.c

```
120 list_pos list_next(const list *l, const list_pos p)
121 {
122     return p->next;
123 }
```

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 *l, 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.
169     e->val = v;
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

```
 9 int *int_create(int v)
10 {
11     int *p = malloc(sizeof(*p)); // allocate payload
12     *p = v; // assign value
13     return p;
14 }
```

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(l);
25     for (int i = 0; i < 2; i++) {
26         int *v = int_create(20 + i);
27         p = list_insert(l, v, p);
28         p = list_next(l, p);
29     }
30     list_print(l, print_int);
```


Hur kodbasen hanterar abstraktion (6)

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

code/list-with-killfunction.c

```
 1 #include <stdio.h>
 2 #include <stdlib.h>
 3 #include <list.h>
 4 void print_int(const void *p)
 5 {
 6     const int *q = p;
 7     printf("%d ", *q);
 8 }
 9 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:
  - `int_create` Allokera dynamiskt minne som rymmer ett heltal, sätt värdet och returnera adressen
  - `int_kill` Lämna tillbaka minnet som tillhör ett heltal
  - `print_int` Tolka innehållet i minnet på en viss adress som ett heltal och skriv ut värdet

Hur kodbasen hanterar abstraktion (7)

- Huvudprogrammet ser ut så här:

code/list-with-killfunction.c

```
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(l);
25     for (int i = 0; i < 2; i++) {
26         int *v = int_create(20 + i);
27         p = list_insert(l, v, p);
28         p = list_next(l, p);
29     }
30     list_print(l, print_int);
31
32
33
34
35
36
37     // Clean up the list, including payload
38     list_kill(l);
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

```
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(l);
25      for (int i = 0; i < 2; i++) {
26          int *v = int_create(20 + i);
27          p = list_insert(l, v, p);
28          p = list_next(l, p);
29      }
30      list_print(l, 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)

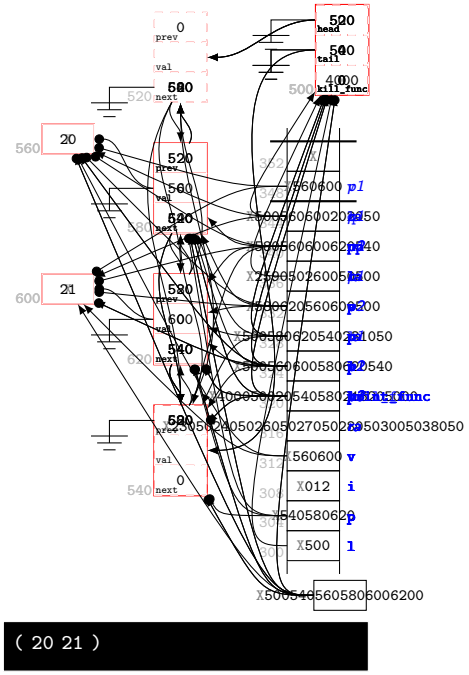
- Slutligen så anropas `list_kill` för att återlämna allt minne som listan använder:

code/list-with-killfunction.c

```
19  int main(void)
20  {
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(l);
25      for (int i = 0; i < 2; i++) {
26          int *v = int_create(20 + i);
27          p = list_insert(l, v, p);
28          p = list_next(l, p);
29      }
30      list_print(l, print_int);
31
32
33
34
35
36
37      // Clean up the list, including payload
38      list_kill(l);
39      return 0;
40  }
```


Exempel:

```
9   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(l);
25      for (int i = 0; i < 2; i++) {
26          int *v = int_create(20 + i);
27          p = list_insert(l, v, p);
28          p = list_next(l, p);
29      }
30      list_print(l, print_int);
31
32
33
34
35
36
37      // Clean up the list, including payload
38      list_kill(l);
39      return 0;
40  }
```




```
( 20 21 )
```


Ansvarsförvirring

Ansvarsförvirring

- Ansvaret för att lämna tillbaka minne som allokerats dynamiskt till listans komponenter (huvud, 2-celler) ligger hos listans funktioner
- Vem har ansvar för att lämna tillbaka minne som allokerats dynamiskt till payload?
- Här finns två alternativ:
  1. Ansvaret för att återlämna minnet behålls av den del av programmet som reserverade minnet
     - Vi kan kalla det för att listan lånat minnet
     - Återlämningen sker i kod som ligger utanför listan
  2. Listan övertar ansvaret för att återlämna minnet
     - Vi kan kalla det för att listan äger minnet
     - Återlämningen sker i kod som tillhör listan

The `kill_function` (1)

- I kodbasen hanteras detta genom att varje generisk datatyp, t.ex. Stack, Lista, etc., har en s.k. `kill_function` kopplad till sig
  - Tilldelningen av en `kill_function` sker vid konstruktionen av datatypen och är kopplat till den instansen av datatypen
  - Det betyder att vissa instanser kan överta ansvaret medan andra gör det inte

The `kill_function` (2)

- Anropet

```
list *l = list_empty(int_kill);
```

  betyder att när ett element tas bort från listan så anropas funktionen `int_kill` med den lagrade pekaren
  - Det motsvarar att listan äger minnet
- Anropet

```
list *l = list_empty(NULL);
```

  betyder att när ett element tas bort från listan så anropas ingen funktion med den lagrade pekaren
  - Det motsvarar att listan lånat minnet

När väljer man vad? (1)

- När väljer man att använda en `kill_function`?

```
list *l = list_empty(int_kill);
```

  - Enklare kod
  - Aktuellt för OU1
  - Lagra en egen kopia av varje värde

När väljer man vad? (2)

- När väljer man att inte använda en `kill_function`?

```
list *l = list_empty(NULL);
```

  - Större kontroll över när elementvärden avallokeras
    - Aktuellt för OU3
  - Om elementvärdena inte är dynamiskt allokerade
  - Gör det möjligt att lagra referenser till värden utan att kopiera dem

När väljer man vad? (3)

- Om vi vill lagra samma värden i flera listor (eller i någon annan datatyp) kan vi kombinera alternativen:
  - En lista äger 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 `kill_function`

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(l);
25     for (int i = 0; i < 2; i++) {
26         int *v = int_create(20 + i);
27         p = list_insert(l, v, p);
28         p = list_next(l, p);
29     }
30     list_print(l, print_int);
31
32
33
34
35
36
37     // Clean up the list, including payload
38     list_kill(l);
39     return 0;
40 }
```

- Motsvarar grovt `list_mwe2.c`

- Lista 3b använder ingen `kill_function`

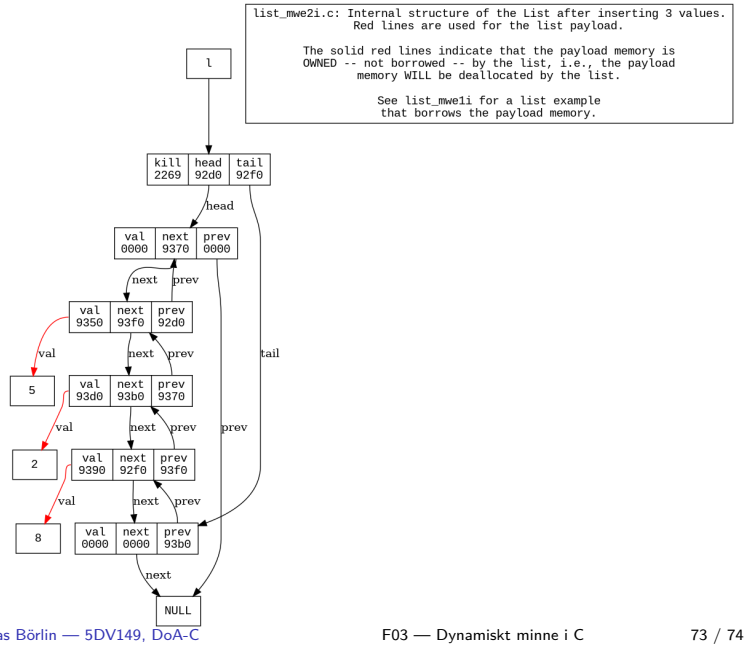
code/list-no-freehandler.c

```
19 int main(void)
20 {
21     // Create empty list, keep responsibility to
22     // deallocate payload
23     list *l = list_empty(NULL);
24     list_pos p = list_first(l);
25     for (int i = 0; i < 2; i++) {
26         int *v = int_create(20 + i);
27         p = list_insert(l, v, p);
28         p = list_next(l, p);
29     }
30     list_print(l, print_int);
31     // Clean up the payloads
32     p = list_first(l);
33     while (!list_pos_is_equal(l, p, list_end(l))) {
34         int *v = list_inspect(l, p);
35         int_kill(v); // free payload
36         p = list_next(l, p);
37     }
38     list_kill(l); // Clean up the list
39     return 0;
40 }
```

- Motsvarar grovt `list_mwe1.c`

Blank

list_mwe2i.c



list_mwe1i.c