



DVA104 VT19

Datastrukturer och Algoritmer
Stefan Bygde



Mer om ADTer och att dölja information

- Kom ihåg vårt Studentexempel från förra föreläsningen.
- Antag att vi istället för att ange studentid:t som parameter till `createNewStudent()` istället vilja automatiskt beräkna det mha namn och startdatum.
- Då behöver vi använda en funktion som gör detta:

```
char* createStudentID(char* name, int startingYear);
```

ADTer och att dölja information

student.c

```
Student createNewStudent(char* name, int startingYear)
{
    assert(name != NULL);
    assert(startingYear > 1977); /* Ingen började innan MDH grundades */

    Student result;
    strcpy(result.name, name);
    result.credits = 0.0;
    strcpy(result.studentID, createStudentID(name, startingYear));

    return result;
}
```

ADTer och att dölja information

Hör createStudentID till interfacet för ADTn Student?

```
char* createStudentID(char* name, int startingYear);
```

ADTer och att dölja information

Hör createStudentID till interfacet för ADTn Student?

```
char* createStudentID(char* name, int startingYear);
```

Svar: **NEJ!**

- Typen Student förekommer varken i argument eller returvärde
- Denna funktion är inte tänkt för “slutanvändaren” att använda
 - Det görs enbart som ett steg i att konstruera en ny student
- Bör dock ändå ligga i en egen funktion

Statiska funktioner

- En statisk funktion är en funktion som bara är tänkt att användas i en fil
- En statisk funktion går inte att använda utanför den fil den är definierad
- Med andra ord: en statisk funktion “finns inte” utanför funktionen -> mindre funktioner att tänka/hålla reda på!
- En statisk funktion läggs EJ i h-filen, eftersom ingen ska se den
- Användbart för:
 - “Hjälpfunktioner”
 - Funktioner som inte är delar av interfacet till en ADT

Statiska funktioner

student.c

```
/* Denna funktion går ej att använda utanför student.c. Den ligger heller ej deklarerad i student.h */
```

```
static char* createStudentID(char* name, int startingYear)
{
    /* ...Kod för att räkna ut student-idt... */
    return studentId;
}
```

```
Student createNewStudent(char* name, int startingYear)
```

```
{
    assert(name != NULL);
    assert(startingYear > 1977); /* Ingen började innan MDH grundades */

    Student result;
    strcpy(result.name, name);
    result.credits = 0.0;
    result.studentID = createStudentID(name, startingYear);
    return result;
}
```

Eftersom vi inte placerar en funktionsdeklaration i h-filen för statiska funktioner är det viktigt att de definieras ovanför anropet till funktionen.

Som konsekvens brukar de statiska funktionerna hamna längst upp i C-filen.

Repetition

- Datorns minne (stacken)
- Funktionsanrop
 - Parametervaribler (argument). Startvärde kommer från den anropande funktionen.
 - Lokala variabler. Värde ges vid deklaration.
 - Parametervariabler och lokala variabler *tas bort* när funktionsanropet är klart. Funktionen återvänder till den anropade funktionen.
 - Tidigare anrop samt dess variabler sparas
- Pekare vid funktionsanrop
 - Gör det möjligt att ändra i variabler från den anropande funktionen.
- Se detaljer i materialet till DVA117

Repetition

```
int h(void)
{
    return 0;
}

int g(void)
{
    return 1+h();
}

int f(void)
{
    return 1+g();
}

int main(void)
{
    int x = 1+f();
    return 0;
}
```

Repetition

```
int h(void)
{
    return 0;
}
```

```
int g(void)
{
    return 1+h();
}
```

```
int f(void)
{
    return 1+g();
}
```

```
int main(void)
{
    int x = 1+f();
    return 0;
}
```

Call stack

main(): x = 1+f()

Repetition

```
int h(void)
{
    return 0;
}
```

```
int g(void)
{
    return 1+h();
}
```

```
int f(void)
{
    return 1+g();
}
```

```
int main(void)
{
    int x = 1+f();
    return 0;
}
```

Call stack

main(): x = 1+f()

f(): return 1+g()

Repetition

```
int h(void)
{
    return 0;
}
```

```
int g(void)
{
    return 1+h();
}
```

```
int f(void)
{
    return 1+g();
}
```

```
int main(void)
{
    int x = 1+f();
    return 0;
}
```

Call stack

main(): x = 1+f()

f(): return 1+g()

g(): return 1+h()

Repetition

```
int h(void)
{
    return 0;
}
```

Return 0

```
int g(void)
{
    return 1+h();
}
```

```
int f(void)
{
    return 1+g();
}
```

```
int main(void)
{
    int x = 1+f();
    return 0;
}
```

Call stack

main(): x = 1+f()

f(): return 1+g()

g(): return 1+h()

h(): return 0



Repetition

```
int h(void)
{
    return 0;
}
```

```
int g(void)
{
    return 1+h();
}
```

```
int f(void)
{
    return 1+g();
}
```

```
int main(void)
{
    int x = 1+f();
    return 0;
}
```

Return 1

Call stack

main(): x = 1+f()

f(): return 1+g()

g(): return 1+0



Repetition

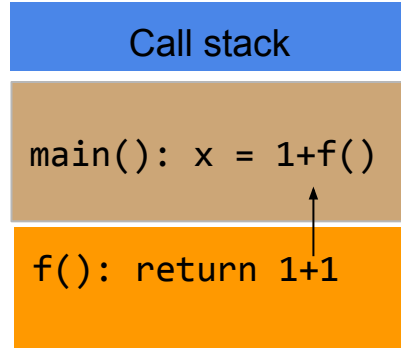
```
int h(void)
{
    return 0;
}
```

```
int g(void)
{
    return 1+h();
}
```

```
int f(void)
{
    return 1+g();
}
```

```
int main(void)
{
    int x = 1+f();
    return 0;
}
```

Return 2



Repetition

```
int h(void)
{
    return 0;
}
```

```
int g(void)
{
    return 1+h();
}
```

```
int f(void)
{
    return 1+g();
}
```

```
int main(void)
{
    int x = 1+f();
    return 0;
}
```

x = 3

Call stack

main(): x=1+2=3

Rekursion

- Rekursion är konceptet att en funktion anropar sig själv. Det vill säga att funktionskroppen till en funktion `f()` innehåller ett anrop till `f()`.
- Exempel:

```
int func(int x, int y)
{
    /* lite kod som gör någonting... */
    if(x==0)
        y = func(10,20); /* Anrop till func(): rekursion! */
}
```

Rekursion

- Varför rekursion?
 - Många problem kan naturligt lösas genom rekursion
 - Rekursion kan ge upphov till väldigt små (lite kod) och eleganta lösningar
 - Rekursion används ofta tillsammans med datatyper (länkade listor, binära träd)
 - Rekursion används i många sök- och sorteringsalgoritmer

Rekursion handlar om att definiera något i termer av sig själv. Vi börjar med att titta på ett exempel på hur vanlig matematisk multiplikation kan definieras i termer av sig själv.

Multiplikation (med heltal)

- Hur räknar man ut $4*5$?
- Vi vet att $4*5 = 5 + 3*5$. Eller hur?
- Vi vet att $3*5 = 5 + 2*5$.
- Vi vet att $2*5 = 5 + 1*5$.
- Vi vet att $1*5 = 5$.

Mer generellt, om m och n är heltal så:

- $m*n = n + (m-1)*n$ (om $m > 1$)
- $m*n = n$ (om $m = 1$)

Vi har alltså definierat
multiplikation mha
multiplikation...

Multiplikation (med heltal)

Med denna definition:

$$4*5$$

$$\begin{array}{ll} m*n = n + (m-1)*n & (\text{om } m > 1) \\ m*n = n & (\text{om } m = 1) \end{array}$$

Vi applicerar nu definitionen rakt av för $m=4$ och $n = 5$. Och får alltså:

$$4*5 = 5 + 3*5$$

Multiplikation (med heltal)

Med denna definition:

$$\begin{aligned} &4*5 \\ &= 5 + (3*5) \end{aligned}$$

Vi fortsätter nu likadant för att beräkna $3*5$.

$$\begin{aligned} m*n &= n + (m-1)*n && (\text{om } m > 1) \\ m*n &= n && (\text{om } m = 1) \end{aligned}$$

$$\begin{aligned} m &= 4 \\ n &= 5 \end{aligned}$$

Multiplikation (med heltal)

Med denna definition:

$$\begin{aligned}4*5 \\&= 5 + (3*5) \\&= 5 + 5 + (2*5)\end{aligned}$$

Samma definition ger
 $3*5 = 5 + (2*5)$

Vi fortsätter att beräkna
 $2*5...$

$$\begin{aligned}m*n &= n + (m-1)*n && (\text{om } m > 1) \\m*n &= n && (\text{om } m = 1)\end{aligned}$$

$$\begin{aligned}m &= 3 \\n &= 5\end{aligned}$$

Multiplikation (med heltal)

Med denna definition:

$$\begin{aligned} &4*5 \\ &= 5 + (3*5) \\ &= 5 + 5 + (2*5) \\ &= 5 + 5 + 5 + (1*5) \end{aligned}$$

$$\begin{aligned} m*n &= n + (m-1)*n && (\text{om } m > 1) \\ m*n &= n && (\text{om } m = 1) \end{aligned}$$

$$\begin{aligned} m &= 2 \\ n &= 5 \end{aligned}$$

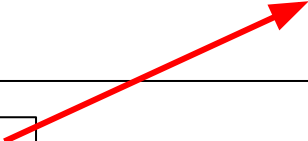
Multiplikation (med heltal)

Med denna definition:

$$\begin{aligned} &4*5 \\ &= 5 + (3*5) \\ &= 5 + 5 + (2*5) \\ &= 5 + 5 + 5 + (1*5) \\ &= 5 + 5 + 5 + 5 \end{aligned}$$

För att beräkna $1*5$
använder vi den andra
delen av definitionen

$$\begin{aligned} m*n &= n + (m-1)*n && (\text{om } m > 1) \\ m*n &= n && (\text{om } m = 1) \end{aligned}$$

$$\begin{aligned} m &= 1 \\ n &= 5 \end{aligned}$$


Multiplikation (med heltal)

Med denna definition:

$$\begin{aligned} &4*5 \\ &= 5 + (3*5) \\ &= 5 + 5 + (2*5) \\ &= 5 + 5 + 5 + (1*5) \\ &= 5 + 5 + 5 + 5 \\ &= 5 + 5 + 10 \end{aligned}$$

$m*n = n + (m-1)*n$	(om $m > 1$)
$m*n = n$	(om $m = 1$)

Multiplikation (med heltal)

Med denna definition:

$$\begin{aligned} &4*5 \\ &= 5 + (3*5) \\ &= 5 + 5 + (2*5) \\ &= 5 + 5 + 5 + (1*5) \\ &= 5 + 5 + 5 + 5 \\ &= 5 + 5 + 10 \\ &= 5 + 15 \end{aligned}$$

$m*n = n + (m-1)*n$	(om $m > 1$)
$m*n = n$	(om $m = 1$)

Multiplikation (med heltal)

Med denna definition:

$$\begin{aligned}4*5 &= 5 + (3*5) \\ &= 5 + 5 + (2*5) \\ &= 5 + 5 + 5 + (1*5) \\ &= 5 + 5 + 5 + 5 \\ &= 5 + 5 + 10 \\ &= 5 + 15 \\ &= 20\end{aligned}$$

$m*n = n + (m-1)*n$	(om $m > 1$)
$m*n = n$	(om $m = 1$)

Multiplikation

I allmänhet, för m och n gäller alltså:

$$m * n = n + (m-1) * n \quad (\text{om } m > 1)$$

$$m * n = n \quad (\text{om } m = 1)$$

Vi kan också utöka med regeln, för att få multiplikation med 0 att fungera:

$$m * n = 0 \quad (\text{om } m = 0)$$

Multiplikation

Vi kan nu använda denna specifikation...

$$m * n = n + (m-1) * n \quad (\text{om } m > 1)$$

$$m * n = n \quad (\text{om } m = 1)$$

$$m * n = 0 \quad (\text{om } m = 0)$$

...mer eller mindre rakt av i C kod för att beräkna multiplikation!

Multiplikation med Rekursion

```
int multiply(int m, int n)
{
    if(m==0)
        return 0;
    if(m==1)
        return n;

    return n + multiply(m-1,y);
}
```

Rekursiv specifikation:

$m * n = n + (m-1) * n$ (om $m > 1$)

$m * n = n$ (om $m = 1$)

$m * n = 0$ (om $m = 0$)

Som du ser följer koden den matematiska definitionen precis.

Vi ser också att funktionskroppen till `multiply()` innehåller ett anrop till `multiply()`!

Det är så vi vet att det är frågan om rekursion.

Multiplikation med Rekursion

```
int multiply(int m, int n)
{
    if(m==0)
        return 0;
    if(m==1)
        return n;

    return n + multiply(m-1,y);
}
```

$$\begin{aligned} &4*5 \\ &= 5 + (3*5) \end{aligned}$$

```
multiply(4,5)
m = 4
n = 5
Return 5 + multiply(3,5)
```

Multiplikation med Rekursion

```
int multiply(int m, int n)
{
    if(m==0)
        return 0;
    if(m==1)
        return n;

    return n + multiply(m-1,n);
}
```

$$\begin{aligned} &4*5 \\ &= 5 + (3*5) \\ &= 5 + 5 + (2*5) \end{aligned}$$

```
multiply(4,5)
m = 4
n = 5
Return 5 + multiply(3,5)
```

```
multiply(3,5)
m = 3
n = 5
Return 5 + multiply(2,5)
```


Multiplikation med Rekursion

```
int multiply(int m, int n)
{
    if(m==0)
        return 0;
    if(m==1)
        return n;

    return n + multiply(m-1,n);
}
```

$$\begin{aligned} &4*5 \\ &= 5 + (3*5) \\ &= 5 + 5 + (2*5) \\ &= 5 + 5 + 5 + (1*5) \end{aligned}$$

```
multiply(4,5)
m = 4
n = 5
Return 5 + multiply(3,5)
```

```
multiply(3,5)
m = 3
n = 5
Return 5 + multiply(2,5)
```

```
multiply(2,5)
m = 2
n = 5
Return 5 + multiply(1,5)
```

Multiplikation med Rekursion

```
int multiply(int m, int n)
{
    if(m==0)
        return 0;
    if(m==1)
        return n;

    return n + multiply(m-1,y);
}
```

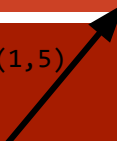
4×5
 $= 5 + (3 \times 5)$
 $= 5 + 5 + (2 \times 5)$
 $= 5 + 5 + 5 + (1 \times 5)$
 $= 5 + 5 + 5 + 5$

```
multiply(4,5)
m = 4
n = 5
Return 5 + multiply(3,5)
```

```
multiply(3,5)
m = 3
n = 5
Return 5 + multiply(2,5)
```

```
multiply(2,5)
m = 2
n = 5
Return 5 + multiply(1,5)
```

```
multiply(1,5)
m = 1
n = 5
Return 5
```



Multiplikation med Rekursion

```
int multiply(int m, int n)
{
    if(m==0)
        return 0;
    if(m==1)
        return n;


    return n + multiply(m-1,y);
}
```

$4 \cdot 5$
 $= 5 + (3 \cdot 5)$
 $= 5 + 5 + (2 \cdot 5)$
 $= 5 + 5 + 5 + (1 \cdot 5)$
 $= 5 + 5 + 5 + 5$
 $= 5 + 5 + 10$

```
multiply(4,5)
m = 4
n = 5
Return 5 + multiply(3,5)
```

```
multiply(3,5)
m = 3
n = 5
Return 5 + multiply(2,5)
```

```
multiply(2,5)
m = 2
n = 5
Return 5 + 5
```



Multiplikation med Rekursion

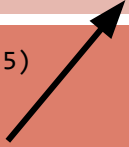
```
int multiply(int m, int n)
{
    if(m==0)
        return 0;
    if(m==1)
        return n;

    return n + multiply(m-1,n);
}
```

4×5
 $= 5 + (3 \times 5)$
 $= 5 + 5 + (2 \times 5)$
 $= 5 + 5 + 5 + (1 \times 5)$
 $= 5 + 5 + 5 + 5$
 $= 5 + 5 + 10$
 $= 5 + 15$

```
multiply(4,5)
m = 4
n = 5
Return 5 + multiply(3,5)
```

```
multiply(3,5)
m = 3
n = 5
Return 5 + 10
```



Multiplikation med Rekursion

```
int multiply(int m, int n)
{
    if(m==0)
        return 0;
    if(m==1)
        return n;

    return n + multiply(m-1,y);
}
```

```
4*5
= 5 + (3*5)
= 5 + 5 + (2*5)
= 5 + 5 + 5 + (1*5)
= 5 + 5 + 5 + 5
= 5 + 5 + 10
= 5 + 15
= 20
```

```
multiply(4,5)
m = 4
n = 5
Return 5 + 15
```

Rekursiva funktioner

- Rekursiva funktioner är alltså ett alternativt sätt att göra en loop!
- Loopar måste ha termineringsvillkor (dvs villkor som gör att loopen tillslut avslutas), detsamma gäller rekursiva funktioner, men i rekursiva funktioner kallas detta för *basfall*.

Rekursiva funktioner

- Alla rekursiva funktioner behöver ett *basfall*
 - Ett basfall är ett fall då ett rekursivt anrop *inte sker*
 - Precis som loopar behöver också något *förändras* i varje anrop för att ett basfall ska uppstå

```
int multiply(int m, int n)
{
    if(m==0)          /* Basfall */
        return 0;
    if(m==1)          /* Basfall */
        return n;

    return n + multiply(m-1,y); /* Rekursivt fall */
}
```

Om vi inte har minst ett basfall så kommer rekursionen inte att terminera!

Basfall, exempel

```
int multiply(int x, int y)
{
    if(x==0)
        return 0; /* Basfall 1: inget rekursivt anrop */
    if(x==1)
        return y; /* Basfall 2: inget rekursivt anrop */

    return y + multiply(x-1,y); /* rekursivt anrop, med SKILLNAD i argumenten */
}
```


Loopar vs rekursion

- Loopar
 - Måste ha minst ett avslutningsvillkor (t.ex **while**(villkor))
 - Något måste förändras i varje varv så att villkoret uppstår (t.ex `i++`)
- Rekursion
 - Måste ha minst ett basfall (t.ex **if**(villkor) **return** 0;)
 - Argumenten måste förändras i varje rekursivt anrop så att basfallet uppstår (t.ex. $f(x-1)$)
- Loopar och rekursion är beräkningsmässigt ekvivalenta, dvs:
 - Allt som går att göra med en loop går att göra med rekursion och vice versa
 - Dock kan det vara olika praktiskt/effektivt

Loopar och rekursion är ekvivalent

```
// Loop
int i;
for(i=0; i<5; i = i+1)
    printf("varv %d\n",i+1);

// Rekursion (anropas med f(0) )
void f(int i)
{
    if(i<5)
    {
        printf("varv %d\n", i+1);
        f(i+1);                // Rekursivt fall
    }
    else
        return; // Basfall
}
```

Loopar och rekursion är ekvivalent

```
// Loop
int i;
for(i=0; i<5; i = i+1)
    printf("varv %d\n",i+1);
```

Blå text: förändring

Röd text: villkor

```
// Rekursion (anropas med f(0) )
void f(int i)
{
    if(i<5)
    {
        printf("varv %d\n", i+1);
        f(i+1);
    }
    else
        return;
}
```

Som du kan se är det samma förändring
och samma termineringsvillkor!

Pekare

```
void allocateArray(int* arr, unsigned int size)
{
    int i;
    arr = (int*)malloc(sizeof(int)*size);

    for(i=0; i<size; i++)
        arr[i] = 10;
}

int main(void)
{
    int* x = NULL;
    allocateArray(x,5);
    printf("%d", *x); /* Vad skrivs ut? */
}
```

Pekare

```
void allocateArray(int* arr, unsigned int size)
{
    int i;
    arr = (int*)malloc(sizeof(int)*size);

    for(i=0; i<size; i++)
        arr[i] = 10;
}

int main(void)
{
    → int* x = NULL;
    allocateArray(x,5);
    printf("%d", *x); /* Vad skrivs ut? */
}
```

main()

1000	[0]	x
1001	[]	
1002	[]	
1003	[]	

Pekare

```
→ void allocateArray(int* arr, unsigned int size)
{
    int i;
    arr = (int*)malloc(sizeof(int)*size);

    for(i=0; i<size; i++)
        arr[i] = 10;
}

int main(void)
{
    int* x = NULL;
    → allocateArray(x,5);
    printf("%d", *x); /* Vad skrivs ut? */
}
```

main()

1000	[0]	x
1001	[]	
1002	[]	
1003	[]	

allocateArray()

1050	[0]	arr
1051	[]	
1052	[]	
1053	[]	
1054	[5]	size
1055	[]	
1056	[]	
1057	[]	

Pekare

```
void allocateArray(int* arr, unsigned int size)
{
    → int i;
      arr = (int*)malloc(sizeof(int)*size);

      for(i=0; i<size; i++)
          arr[i] = 10;
}

int main(void)
{
    int* x = NULL;
    → allocateArray(x,5);
      printf("%d", *x); /* Vad skrivs ut? */
}
```

main()

```
1000 [ 0 ] x
1001 [   ]
1002 [   ]
1003 [   ]
```

allocateArray()

```
1050 [ 0 ] arr
1051 [   ]
1052 [   ]
1053 [   ]
1054 [ 5 ] size
1055 [   ]
1056 [   ]
1057 [   ]
1058 [????] i
1059 [????]
1060 [????]
1061 [????]
```

Pekare

```
void allocateArray(int* arr, unsigned int size)
{
    int i;
    → arr = (int*)malloc(sizeof(int)*size);

    for(i=0; i<size; i++)
        arr[i] = 10;
}

int main(void)
{
    int* x = NULL;
    → allocateArray(x,5);
    printf("%d", *x); /* Vad skrivs ut? */
}
```

main()

1000	[0]	x
1001	[]	
1002	[]	
1003	[]	

allocateArray()

1050	[5000]	arr
1051	[]	
1052	[]	
1053	[]	
1054	[5]	size
1055	[]	
1056	[]	
1057	[]	
1058	[????]	i
1059	[????]	
1060	[????]	
1061	[????]	

Heap

5000	[?]	arr[0]
5001	[?]	
5002	[?]	
5003	[?]	
5004	[?]	arr[1]
5005	[?]	
5006	[?]	
5007	[?]	
5008	[?]	arr[2]
5009	[?]	
5010	[?]	
5011	[?]	
5012	[?]	arr[3]
5013	[?]	
5014	[?]	
5015	[?]	
5016	[?]	arr[4]
5017	[?]	
5018	[?]	
5019	[?]	

Pekare

```
void allocateArray(int* arr, unsigned int size)
{
    int i;
    arr = (int*)malloc(sizeof(int)*size);

    → for(i=0; i<size; i++)
        arr[i] = 10;
}

int main(void)
{
    int* x = NULL;
    → allocateArray(x,5);
    printf("%d", *x); /* Vad skrivs ut? */
}
```

main()

1000	[0]	x
1001	[]	
1002	[]	
1003	[]	

allocateArray()

1050	[5000]	arr
1051	[]	
1052	[]	
1053	[]	
1054	[5]	size
1055	[]	
1056	[]	
1057	[]	
1058	[0]	i
1059	[]	
1060	[]	
1061	[]	

Heap

5000	[?]	arr[0]
5001	[?]	
5002	[?]	
5003	[?]	
5004	[?]	arr[1]
5005	[?]	
5006	[?]	
5007	[?]	
5008	[?]	arr[2]
5009	[?]	
5010	[?]	
5011	[?]	
5012	[?]	arr[3]
5013	[?]	
5014	[?]	
5015	[?]	
5016	[?]	arr[4]
5017	[?]	
5018	[?]	
5019	[?]	

Pekare

```
void allocateArray(int* arr, unsigned int size)
{
    int i;
    arr = (int*)malloc(sizeof(int)*size);

    for(i=0; i<size; i++)
        → arr[i] = 10;
}

int main(void)
{
    int* x = NULL;
    → allocateArray(x,5);
    printf("%d", *x); /* Vad skrivs ut? */
}
```

main()

1000	[0]	x
1001	[]	
1002	[]	
1003	[]	

allocateArray()

1050	[5000]	arr
1051	[]	
1052	[]	
1053	[]	
1054	[5]	size
1055	[]	
1056	[]	
1057	[]	
1058	[0]	i
1059	[]	
1060	[]	
1061	[]	

Heap

5000	[10]	arr[0]
5001	[]	
5002	[]	
5003	[]	
5004	[?]	arr[1]
5005	[?]	
5006	[?]	
5007	[?]	
5008	[?]	arr[2]
5009	[?]	
5010	[?]	
5011	[?]	
5012	[?]	arr[3]
5013	[?]	
5014	[?]	
5015	[?]	
5016	[?]	arr[4]
5017	[?]	
5018	[?]	
5019	[?]	

Pekare

```
void allocateArray(int* arr, unsigned int size)
{
    int i;
    arr = (int*)malloc(sizeof(int)*size);

    → for(i=0; i<size; i++)
        arr[i] = 10;
}

int main(void)
{
    int* x = NULL;
    → allocateArray(x,5);
    printf("%d", *x); /* Vad skrivs ut? */
}
```

Efter 5 varv!

main()

1000	[0]	x
1001	[]	
1002	[]	
1003	[]	

allocateArray()

1050	[5000]	arr
1051	[]	
1052	[]	
1053	[]	
1054	[5]	size
1055	[]	
1056	[]	
1057	[]	
1058	[5]	i
1059	[]	
1060	[]	
1061	[]	

Heap

5000	[10]	arr[0]
5001	[]	
5002	[]	
5003	[]	
5004	[10]	arr[1]
5005	[]	
5006	[]	
5007	[]	
5008	[10]	arr[2]
5009	[]	
5010	[]	
5011	[]	
5012	[10]	arr[3]
5013	[]	
5014	[]	
5015	[]	
5016	[10]	arr[4]
5017	[]	
5018	[]	
5019	[]	

Pekare

```
void allocateArray(int* arr, unsigned int size)
{
    int i;
    arr = (int*)malloc(sizeof(int)*size);

    for(i=0; i<size; i++)
        arr[i] = 10;
→ }
```

```
int main(void)
{
    int* x = NULL;
→ allocateArray(x,5);
    printf("%d", *x); /* Vad skrivs ut? */
}
```

main()

1000	[0]	x
1001	[]	
1002	[]	
1003	[]	

Heap

5000	[10]
5001	[]
5002	[]
5003	[]
5004	[10]
5005	[]
5006	[]
5007	[]
5008	[10]
5009	[]
5010	[]
5011	[]
5012	[10]
5013	[]
5014	[]
5015	[]
5016	[10]
5017	[]
5018	[]
5019	[]

Pekare

```
void allocateArray(int* arr, unsigned int size)
{
    int i;
    arr = (int*)malloc(sizeof(int)*size);

    for(i=0; i<size; i++)
        arr[i] = 10;
}

int main(void)
{
    int* x = NULL;
    allocateArray(x,5);
    ➔ printf("%d", *x); /* Vad skrivs ut? */
}
```

main()

1000	[0]	x
1001	[]	
1002	[]	
1003	[]	

Heap

5000	[10]
5001	[]
5002	[]
5003	[]
5004	[10]
5005	[]
5006	[]
5007	[]
5008	[10]
5009	[]
5010	[]
5011	[]
5012	[10]
5013	[]
5014	[]
5015	[]
5016	[10]
5017	[]
5018	[]
5019	[]

Pekare

```
void allocateArray(int* arr, unsigned int size)
{
    int i;
    arr = (int*)malloc(sizeof(int)*size);

    for(i=0; i<size; i++)
        arr[i] = 10;
}

int main(void)
{
    int* x = NULL;
    allocateArray(x,5);
    → printf("%d", *x); /* Vad skrivs ut? */
}
```

main()

1000	[0]	x
1001	[]	
1002	[]	
1003	[]	

Vad skrevs ut?

Vad händer med
heap-minnet?

Heap

5000	[10]
5001	[]
5002	[]
5003	[]
5004	[10]
5005	[]
5006	[]
5007	[]
5008	[10]
5009	[]
5010	[]
5011	[]
5012	[10]
5013	[]
5014	[]
5015	[]
5016	[10]
5017	[]
5018	[]
5019	[]

Pekare

```
void allocateArray(int* arr, unsigned int size)
{
    int i;
    arr = (int*)malloc(sizeof(int)*size);

    for(i=0; i<size; i++)
        arr[i] = 10;
}
```

```
int main(void)
{
    int* x = NULL;
    allocateArray(x,5);
    → printf("%d", *x); /* Vad skrivs ut? */
}
```

main()

1000	[0]	x
1001	[]	
1002	[]	
1003	[]	

Vad skrivs ut?

Svar: x är NULL

*x == *NULL == *0 == KRASH!

Vad händer med
heap-minnet?

Svar: stannar kvar i minnet.

Ingen känner till adressen
5000 och ingen kan
komma åt det.

Heap

5000	[10]
5001	[]
5002	[]
5003	[]
5004	[10]
5005	[]
5006	[]
5007	[]
5008	[10]
5009	[]
5010	[]
5011	[]
5012	[10]
5013	[]
5014	[]
5015	[]
5016	[10]
5017	[]
5018	[]
5019	[]

Pekare: Alternativ lösning

```
int* allocateArray(unsigned int size)
{
    int i;
    int* arr = (int*)malloc(sizeof(int)*size);

    for(i=0; i<size; i++)
        arr[i] = 10;
    return arr;
}

int main(void)
{
    int* x = NULL;
    x = allocateArray(5);
    printf("%d", *x); /* Vad skrivs ut? */
}
```


Pekare: Alternativ lösning

```
int* allocateArray(unsigned int size)
{
    int i;
    int* arr = (int*)malloc(sizeof(int)*size);

    for(i=0; i<size; i++)
        arr[i] = 10;
    return arr;
}

int main(void)
{
    int* x = NULL;
    x = allocateArray(5);
    printf("%d", *x); /* Vad skrivs ut? */
}
```

Problem med denna lösning (i en ADT):

- Vi kan inte använda returvärdet till annat
- Vi behöver komma ihåg att ta emot returvärdet, vilket kan leda till fel som är svåra att upptäcka (om man glömmer). Kompilatorn varnar ej om detta!
- Vi behöver bara ta emot returvärdet från VISSA operationer (de som ändrar värdet). Igen kompilatorn talar inte om.

Pekare: Nytt försök

```
void allocateArray(int** arr, unsigned int size)
{
    int i;
    *arr = (int*)malloc(sizeof(int)*size);

    for(i=0; i<size; i++)
        (*arr)[i] = 10;
}

int main(void)
{
    int* x = NULL;
    allocateArray(&x,5);
    printf("%d", *x); /* Vad skrivs ut? */
}
```

Pekare: Nytt försök

```
void allocateArray(int** arr, unsigned int size)
{
    int i;
    *arr = (int*)malloc(sizeof(int)*size);

    for(i=0; i<size; i++)
        (*arr)[i] = 10;
}

int main(void)
{
    → int* x = NULL;
    allocateArray(&x,5);
    printf("%d", *x); /* Vad skrivs ut? */
}
```

main()

1000	[0]	x
1001	[]	
1002	[]	
1003	[]	

Pekare: Nytt försök

```
→ void allocateArray(int** arr, unsigned int size)
{
    int i;
    *arr = (int*)malloc(sizeof(int)*size);

    for(i=0; i<size; i++)
        (*arr)[i] = 10;
}

int main(void)
{
    int* x = NULL;
    → allocateArray(&x,5);
    printf("%d", *x); /* Vad skrivs ut? */
}
```

main()

1000	[0]	x
1001	[]	
1002	[]	
1003	[]	

allocateArray()

1050	[1000]	arr
1051	[]	
1052	[]	
1053	[]	
1054	[5]	size
1055	[]	
1056	[]	
1057	[]	

Pekare: Nytt försök

```
void allocateArray(int** arr, unsigned int size)
{
    → int i;
    *arr = (int*)malloc(sizeof(int)*size);

    for(i=0; i<size; i++)
        (*arr)[i] = 10;
}

int main(void)
{
    int* x = NULL;
    → allocateArray(&x,5);
    printf("%d", *x); /* Vad skrivs ut? */
}
```

main()

1000	[0]	x
1001	[]	
1002	[]	
1003	[]	

allocateArray()

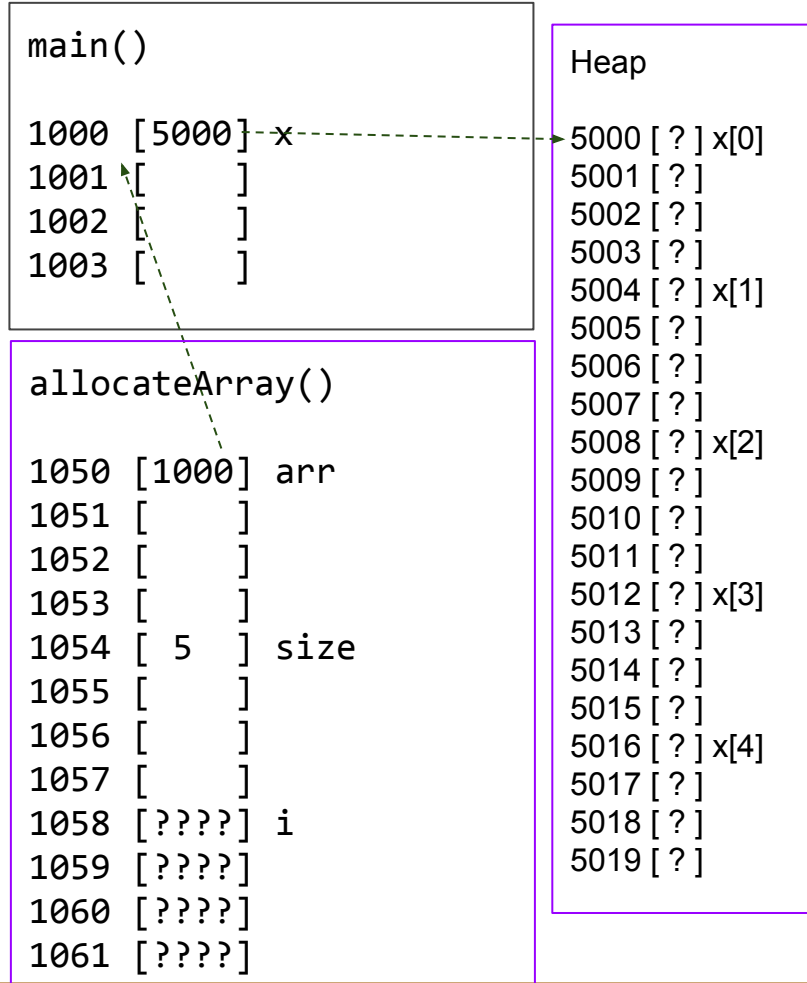
1050	[1000]	arr
1051	[]	
1052	[]	
1053	[]	
1054	[5]	size
1055	[]	
1056	[]	
1057	[]	
1058	[????]	i
1059	[????]	
1060	[????]	
1061	[????]	

Pekare: Nytt försök

```
void allocateArray(int** arr, unsigned int size)
{
    int i;
    → *arr = (int*)malloc(sizeof(int)*size);

    for(i=0; i<size; i++)
        (*arr)[i] = 10;
}

int main(void)
{
    int* x = NULL;
    → allocateArray(&x,5);
    printf("%d", *x); /* Vad skrivs ut? */
}
```

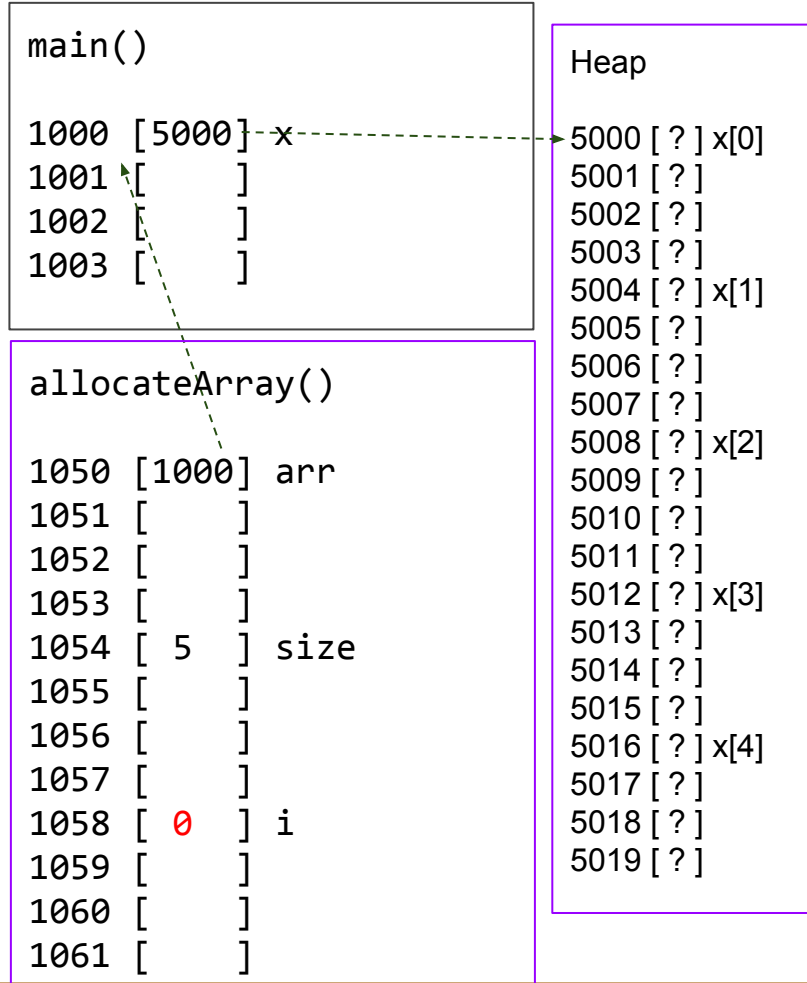


Pekare: Nytt försök

```
void allocateArray(int** arr, unsigned int size)
{
    int i;
    *arr = (int*)malloc(sizeof(int)*size);

    → for(i=0; i<size; i++)
        (*arr)[i] = 10;
}

int main(void)
{
    int* x = NULL;
    → allocateArray(&x,5);
    printf("%d", *x); /* Vad skrivs ut? */
}
```

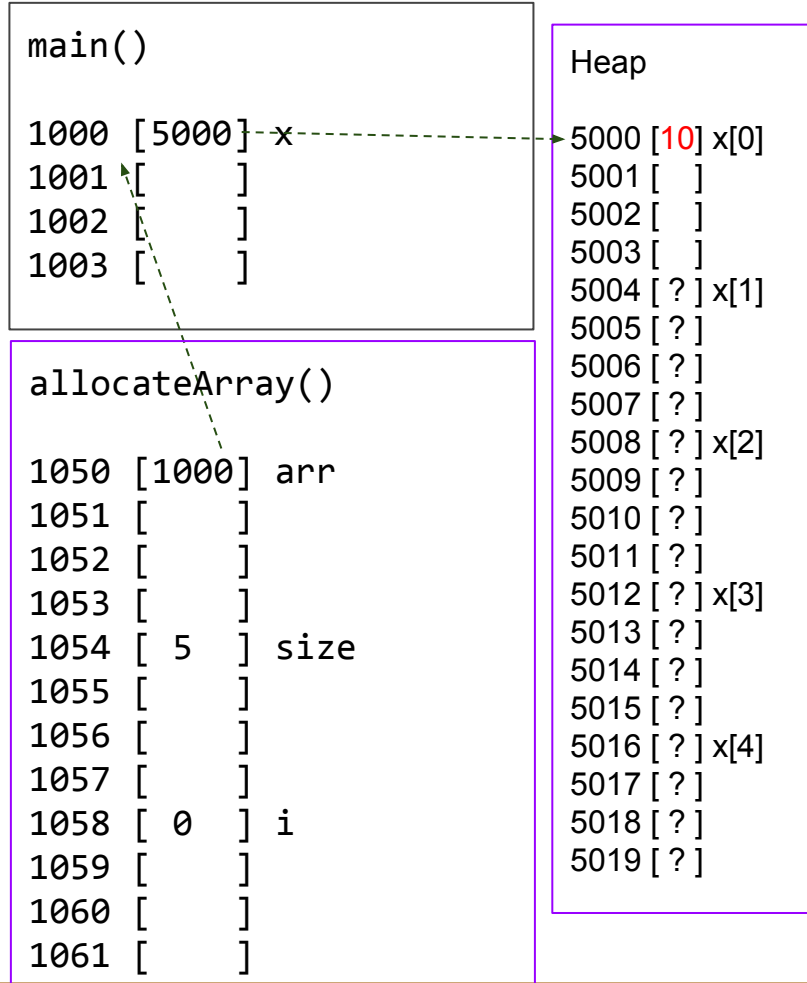


Pekare: Nytt försök

```
void allocateArray(int** arr, unsigned int size)
{
    int i;
    *arr = (int*)malloc(sizeof(int)*size);

    for(i=0; i<size; i++)
        → (*arr)[i] = 10;
}

int main(void)
{
    int* x = NULL;
    → allocateArray(&x,5);
    printf("%d", *x); /* Vad skrivs ut? */
}
```



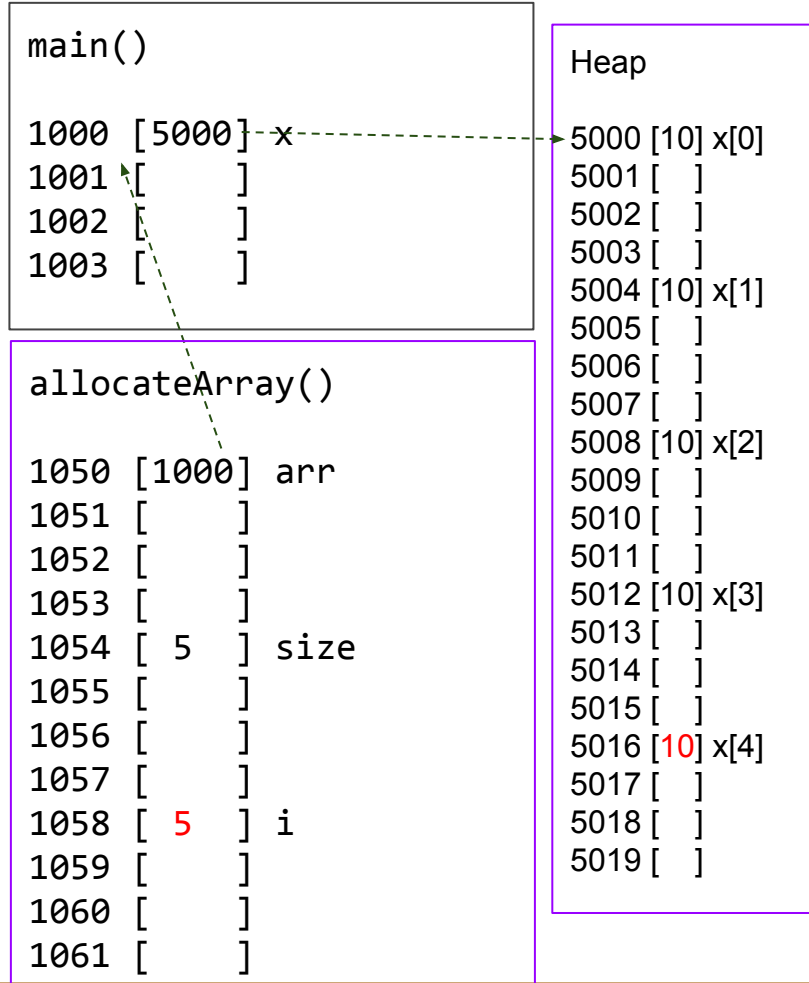
Pekare: Nytt försök

```
void allocateArray(int** arr, unsigned int size)
{
    int i;
    *arr = (int*)malloc(sizeof(int)*size);

    → for(i=0; i<size; i++)
        (*arr)[i] = 10;
}

int main(void)
{
    int* x = NULL;
    → allocateArray(&x,5);
    printf("%d", *x); /* Vad skrivs ut? */
}
```

Efter 5 varv!



Pekare: Nytt försök

```
void allocateArray(int** arr, unsigned int size)
{
    int i;
    *arr = (int*)malloc(sizeof(int)*size);

    for(i=0; i<size; i++)
        (*arr)[i] = 10;
    → }
```

```
int main(void)
{
    int* x = NULL;
    → allocateArray(&x,5);
    printf("%d", *x); /* Vad skrivs ut? */
}
```

main()

1000	[5000]	x
1001	[]	
1002	[]	
1003	[]	

Heap

5000	[10]	x[0]
5001	[]	
5002	[]	
5003	[]	
5004	[10]	x[1]
5005	[]	
5006	[]	
5007	[]	
5008	[10]	x[2]
5009	[]	
5010	[]	
5011	[]	
5012	[10]	x[3]
5013	[]	
5014	[]	
5015	[]	
5016	[10]	x[4]
5017	[]	
5018	[]	
5019	[]	

Pekare: Nytt försök

```
void allocateArray(int** arr, unsigned int size)
{
    int i;
    *arr = (int*)malloc(sizeof(int)*size);

    for(i=0; i<size; i++)
        (*arr)[i] = 10;
}

int main(void)
{
    int* x = NULL;
    allocateArray(&x,5);
    → printf("%d", *x); /* Vad skrivs ut? */
}
```

main()

1000	[5000]	x
1001	[]	
1002	[]	
1003	[]	

Heap

5000	[10]	x[0]
5001	[]	
5002	[]	
5003	[]	
5004	[10]	x[1]
5005	[]	
5006	[]	
5007	[]	
5008	[10]	x[2]
5009	[]	
5010	[]	
5011	[]	
5012	[10]	x[3]
5013	[]	
5014	[]	
5015	[]	
5016	[10]	x[4]
5017	[]	
5018	[]	
5019	[]	

Pekare: Nytt försök

```
void allocateArray(int** arr, unsigned int size)
{
    int i;
    *arr = (int*)malloc(sizeof(int)*size);

    for(i=0; i<size; i++)
        (*arr)[i] = 10;
}

int main(void)
{
    int* x = NULL;
    allocateArray(&x,5);
    → printf("%d", *x); /* Vad skrivs ut? */
}
```

main()

1000	[5000]	x
1001	[]	
1002	[]	
1003	[]	

Vad skrevs ut?
Svar: x är NULL
*x == *5000 == 10!

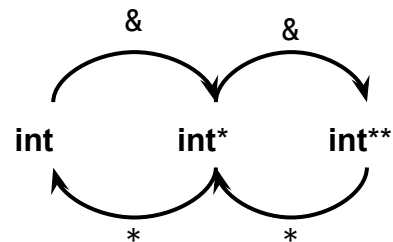
Vi har fortfarande en
pekare till
heap-minnet. Inget
minnesläckage!

Heap

5000	[10]	x[0]
5001	[]	
5002	[]	
5003	[]	
5004	[10]	x[1]
5005	[]	
5006	[]	
5007	[]	
5008	[10]	x[2]
5009	[]	
5010	[]	
5011	[]	
5012	[10]	x[3]
5013	[]	
5014	[]	
5015	[]	
5016	[10]	x[4]
5017	[]	
5018	[]	
5019	[]	

Dubbelppekare

1000 [5] x
1004 [1000] px
1008 [1004] ppx



```
int x = 5;
```

```
int* px = &x;
```

```
int** ppx = &px;
```

- *px
- **px
- &px
- *ppx
- **ppx
- &x
- &ppx

Adress	1000	1004	1008
Värde	5	1000	1004
typ	int	int*	int**
Namn på värdet	5 x *px **ppx	1000 &x px *ppx	1004 &px ppx

Dubbelppekare

1000 [5] x
1004 [1000] px
1008 [1004] ppx

```
int x = 5;  
int* px = &x;  
int** ppx = &px;
```

Vad har följande för typ och för värden?

- *px [typ: int, värde: 5]
- **px [typ: fel, värde: fel]
- &px [typ: int**, värde 1004]
- *ppx [typ: int*, värde 1000]
- **ppx [typ: int, värde 5] (**ppx == *px == x == 5)
- &x [typ: int*, värde 1000] (&x == px)
- &ppx [typ: int***, värde 1008]