# INF-1100: Stack + rekursjon

Einar Holsbø, UiT – Norges arktiske universitet

H23

# Oversikt

- Datastrukturen stack (eng.): hva er det og hvordan brukes det?
- Stacken, (aka. “the call stack”): hvordan gjøres funksjonskall?
- Rekursjon: Hva er det og hvordan kan vi løse problemer med det?

*Stack* er engelsk for *stabel*

Figure 1: En stabel av tallerkener på et hemmelig sted i Tromsø

- Vi stabler "nederst til øverst"
- Vi tar ting ut "øverst til nederst"
- En stabel av tallerkener er **Last In, First Out (LIFO)**

# En *stack* er en LIFO datastruktur (tegning)

- Å legge noe inn kalles en **push**
- Å ta noe ut kalles en **pop**

Aktivitet: fullfør programmet https://bit.ly/3uyOYG1 (løsningsforslag https://bit.ly/3l2kzg7)

```c
#include <stdio.h>

char STACK[128];                // holds stack data
int top = 0;                    // points to top of stack

void push(char ch) {            // implement me
}

char pop() {                    // implement me
}

int main() {
    push(1); push(2); push(3);
  // should print 3 2 1
  printf("%d ", pop()); printf("%d ", pop()); printf("%d\n", pop());
}
```

# Løsningsforslag https://bit.ly/3l2kzg7

```
void push(char ch) {
    STACK[top] = ch;          // place ch on top
    top = top + 1;            // increment pointer
}

char pop() {
    top = top - 1;            // decrement pointer
    return STACK[top];        // return item (will be overwritten on next
}
```

# Eksempel: streng i revers https://bit.ly/3B3l8eV

```
char hello[] = "Helloworld";

int i = 0;

// push until terminating zero
while (hello[i] != 0) {
  push(hello[i]);
  i ++;
}

// match number of pop with number of push!
for (int j = 0; j < i; j ++) {
  hello[j] = pop();
}

printf("%s\n", hello);      // prints dlrowolleH
```

## The Call Stack (stacken)

# Funksjoner trenger plass i minne til argumenter, returverdi, og arbeidsvariabler

```
char subtract(char a, char b) {
  return a - b;
}
```

To muligheter:

# Funksjoner trenger plass i minne til argumenter, returverdi, og arbeidsvariabler

```
char subtract(char a, char b) {
  return a - b;
}
```

To muligheter:

**1) Sett av en fast plass i minnet til hver funkjson**

- Lett å implementere
- Funksjoner kan aldri kalle seg selv (hvorfor?)

# Funksjoner trenger plass i minne til argumenter, returverdi, og arbeidsvariabler

```
char subtract(char a, char b) {
  return a - b;
}
```

To muligheter:

**1) Sett av en fast plass i minnet til hver funkjson**

- Lett å implementere
- Funksjoner kan aldri kalle seg selv (hvorfor?)

**2) Sett av en bit med minne for hvert **kall** til en funksjon**

- Enkelt hvis man bruker en stack
- Funksjoner kan kalle seg selv

## Einar-style assembly for kallet til `subtract` + tegning

C:

```c
char diff = subtract(6, 5);
```

E.-s. assembly:

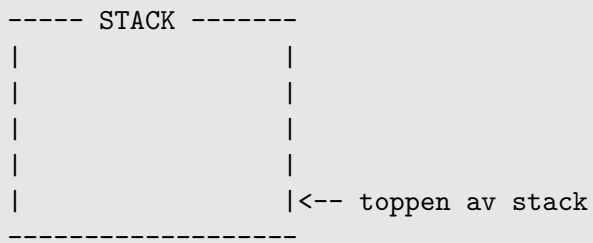
```
    PUSH 5                  // put arguments on stack
    PUSH 6
    CALL subtract           // jump to subtract code
    POP r1                  // pop return value into register 1
    ...
# subtract:
    POP r2                  // pop argument into register 2
    POP r3                  // pop argument into register 3
    SUB $(r3), $(r2), r2    // subtract value in r2 from r3, store in r2
    PUSH $(r2)              // push the result in r2
    RETURN                  // jumps back via return ptr
```

# Noen funksjoner som kaller hverandre

Psevdokode:

```
int a(int arg) { ...; ret_b = b(arg); ...;  }
int b(int arg) { ...; ret_c = c(arg); ...;  }
```

Call Stack



# Noen funksjoner som kaller hverandre

Psevdokode:

```
int a(int arg) { ...; ret_b = b(arg); ...;  }
int b(int arg) { ...; ret_c = c(arg); ...;  }
```

Call Stack

```
Vi kaller a( ... )


----- STACK -------
|                  |
|                  |
|                  |
|                  |
|                  |<-- toppen av stack
--------------------
```

# Noen funksjoner som kaller hverandre

Psevdokode:

```
int a(int arg) { ...; ret_b = b(arg); ...;  }
int b(int arg) { ...; ret_c = c(arg); ...;  }
```

Call Stack

```
Vi kaller a( ... )


----- STACK -------
|                  |
|                  |
|                  |
|                  |<-- toppen av stack
| a() STACK FRAME  |
--------------------
```

# Noen funksjoner som kaller hverandre

Psevdokode:

```
int a(int arg) { ...; ret_b = b(arg); ...;  }
int b(int arg) { ...; ret_c = c(arg); ...;  }
```

Call Stack

```
a( ... ) kaller på b( ... )


----- STACK -------
|                  |
|                  |
|                  |
|                  |<-- toppen av stack
| a() STACK FRAME  |
--------------------
```

# Noen funksjoner som kaller hverandre

Psevdokode:

```
int a(int arg) { ...; ret_b = b(arg); ...;  }
int b(int arg) { ...; ret_c = c(arg); ...;  }
```

Call Stack

```
a( ... ) kaller på b( ... )


----- STACK -------
|                  |
|                  |
|                  |<-- toppen av stack
| b() STACK FRAME  |
| a() STACK FRAME  |
--------------------
```

# Noen funksjoner som kaller hverandre

Psevdokode:

```
int a(int arg) { ...; ret_b = b(arg); ...;  }
int b(int arg) { ...; ret_c = c(arg); ...;  }
```

Call Stack

```
b( ... ) kaller på c( ... )


----- STACK -------
|                  |
|                  |
|                  |<-- toppen av stack
| b() STACK FRAME  |
| a() STACK FRAME  |
--------------------
```

# Noen funksjoner som kaller hverandre

Psevdokode:

```
int a(int arg) { ...; ret_b = b(arg); ...;  }
int b(int arg) { ...; ret_c = c(arg); ...;  }
```

Call Stack

```
b( ... ) kaller på c( ... )


----- STACK -------
|                  |
|                  |<-- toppen av stack
| c() STACK FRAME  |
| b() STACK FRAME  |
| a() STACK FRAME  |
--------------------
```

# Noen funksjoner som kaller hverandre

Psevdokode:

```
int a(int arg) { ...; ret_b = b(arg); ...;  }
int b(int arg) { ...; ret_c = c(arg); ...;  }
```

Call Stack

```
c( ... ) returnerer


----- STACK -------
|                  |
|                  |<-- toppen av stack
| c() STACK FRAME  |
| b() STACK FRAME  |
| a() STACK FRAME  |
--------------------
```

# Noen funksjoner som kaller hverandre

Psevdokode:

```
int a(int arg) { ...; ret_b = b(arg); ...;  }
int b(int arg) { ...; ret_c = c(arg); ...;  }
```

Call Stack

```
c( ... ) returnerer


----- STACK -------
|                  |
|                  |
|                  |<-- toppen av stack
| b() STACK FRAME  |
| a() STACK FRAME  |
--------------------
```

# Noen funksjoner som kaller hverandre

Psevdokode:

```
int a(int arg) { ...; ret_b = b(arg); ...;  }
int b(int arg) { ...; ret_c = c(arg); ...;  }
```

Call Stack

```
b( ... ) returnerer


----- STACK -------
|                  |
|                  |
|                  |<-- toppen av stack
| b() STACK FRAME  |
| a() STACK FRAME  |
--------------------
```

# Noen funksjoner som kaller hverandre

Psevdokode:

```
int a(int arg) { ...; ret_b = b(arg); ...;  }
int b(int arg) { ...; ret_c = c(arg); ...;  }
```

Call Stack

```
b( ... ) returnerer


----- STACK -------
|                  |
|                  |
|                  |
|                  |<-- toppen av stack
| a() STACK FRAME  |
--------------------
```

# Noen funksjoner som kaller hverandre

Psevdokode:

```
int a(int arg) { ...; ret_b = b(arg); ...;  }
int b(int arg) { ...; ret_c = c(arg); ...;  }
```

Call Stack

```
a( ... ) returnerer


----- STACK -------
|                  |
|                  |
|                  |
|                  |<-- toppen av stack
| a() STACK FRAME  |
--------------------
```

# Noen funksjoner som kaller hverandre

Psevdokode:

```
int a(int arg) { ...; ret_b = b(arg); ...;  }
int b(int arg) { ...; ret_c = c(arg); ...;  }
```

Call Stack

```
a( ... ) returnerer


----- STACK -------
|                  |
|                  |
|                  |
|                  |
|                  |<-- toppen av stack
--------------------
```

# Noen funksjoner som kaller hverandre

## Psevdokode:

```
int a(int arg) { ...; ret_b = b(arg); ...;  }
int b(int arg) { ...; ret_c = c(arg); ...;  }
```

## Call Stack

Stack frame blir plassen i minnet
som brukes til funksjonens data:
allokeres for hvert funksjonskall

# Tegning igjen: minneallokering + peker

```c
int main() {
    int *result;

    // stuff happpens...

    result = malloc(sizeof(int));

    // ...

    *result = 12345;

    // ...

    return 0;
}
```

# Aktivitet: funksjonskall med stack

Den nederste koden bruker stacken fra tidligere til å returnere resultatet. Endre koden til også å bruke stacken til argumentene!

## Vanlig funksjonskall

```
char subtract(char a, char b) {
  return a - b;
}
```

## Med stack https://bit.ly/3B0WJXz (løsningsforslag https://bit.ly/2YplKxo)

```
void subtract(char a, char b) {
  push(a - b);
}
int main() {
  subtract(6, 5);
  printf("6 - 5 = %d\n", pop());
}
```

# Løsningsforslag https://bit.ly/2YplKxo

```
void subtract() {
    char a = pop();
    char b = pop();
    push(a - b);
}

// i main blir det (har ikke plass til alt på slide):
    push(5);
    push(6);
    subtract();
    printf("6 - 5 = %d\n", pop());
```

```
    PUSH 5                  // put arguments on stack
    PUSH 6
    CALL subtract           // jump to subtract code
    POP r1                  // pop return value into register 1
```

## Rekursjon

# En rekursiv funksjon er en som kaller seg selv

Fakultetsfunksjonen $n! = n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1$

# En rekursiv funksjon er en som kaller seg selv

Fakultetsfunksjonen $n! = n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1$

Eller: $n! = n \times (n-1)!$, med $1! = 1$ (viktig).

# En rekursiv funksjon er en som kaller seg selv

Fakultetsfunksjonen $n! = n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1$

Eller: $n! = n \times (n-1)!$, med $1! = 1$ (viktig).

```
int factorial(n) {
  if (n == 1) return 1;
  return n*factorial(n-1);
}
```

# En rekursiv funksjon er en som kaller seg selv

Fakultetsfunksjonen $n! = n \times (n-1) \times (n-2) \times \ldots \times 2 \times 1$

Eller: $n! = n \times (n-1)!$, med $1! = 1$ (viktig).

```
int factorial(n) {
  if (n == 1) return 1;
  return n*factorial(n-1);
}
```

Merk: `if`-test i starten (viktig)

# Call stack for factorial(4)

```
int factorial(n) {
  if (n == 1) return 1;
  return n*factorial(n-1);
}
```

```
factorial(4)

----- STACK -------
|                  |
|                  |
|                  |
|                  |
|                  |<-- toppen av stack
--------------------
```

# Call stack for factorial(4)

```
int factorial(n) {
  if (n == 1) return 1;
  return n*factorial(n-1);
}
```

```
factorial(4)

----- STACK -------
|                  |
|                  |
|                  |
|                  |<-- toppen av stack
|  factorial(4)    |
--------------------
```

# Call stack for factorial(4)

```
int factorial(n) {
  if (n == 1) return 1;
  return n*factorial(n-1);
}
```

```
factorial(3)

----- STACK -------
|                  |
|                  |
|                  |
|                  |<-- toppen av stack
|  factorial(4)    |
--------------------
```

# Call stack for factorial(4)

```
int factorial(n) {
  if (n == 1) return 1;
  return n*factorial(n-1);
}
```

```
factorial(3)

----- STACK -------
|                  |
|                  |
|                  |<-- toppen av stack
|  factorial(3)    |
|  factorial(4)    |
--------------------
```

# Call stack for factorial(4)

```
int factorial(n) {
  if (n == 1) return 1;
  return n*factorial(n-1);
}
```

```
factorial(2)

----- STACK -------
|                  |
|                  |
|                  |<-- toppen av stack
|  factorial(3)    |
|  factorial(4)    |
--------------------
```

# Call stack for factorial(4)

```
int factorial(n) {
  if (n == 1) return 1;
  return n*factorial(n-1);
}
```

```
factorial(2)

----- STACK -------
|                  |
|                  |<-- toppen av stack
|  factorial(2)    |
|  factorial(3)    |
|  factorial(4)    |
--------------------
```

# Call stack for factorial(4)

```
int factorial(n) {
  if (n == 1) return 1;
  return n*factorial(n-1);
}
```

```
factorial(1)

----- STACK -------
|                  |
|                  |<-- toppen av stack
|  factorial(2)    |
|  factorial(3)    |
|  factorial(4)    |
--------------------
```

# Call stack for factorial(4)

```
int factorial(n) {
  if (n == 1) return 1;
  return n*factorial(n-1);
}
```

```
factorial(1)

----- STACK -------
|                 |<-- toppen av stack
|  factorial(1)   |
|  factorial(2)   |
|  factorial(3)   |
|  factorial(4)   |
-------------------
```

# Call stack for factorial(4)

```
int factorial(n) {
  if (n == 1) return 1;          //// NB!!!!!!!!!!!!
  return n*factorial(n-1);
}
```

```
factorial(1) returnerer 1

----- STACK -------
|                 |<-- toppen av stack
|  factorial(1)   |
|  factorial(2)   |
|  factorial(3)   |
|  factorial(4)   |
-------------------
```

# Call stack for factorial(4)

```
int factorial(n) {
  if (n == 1) return 1;          //// NB!!!!!!!!!!!!
  return n*factorial(n-1);
}
```

```
factorial(2) returnerer 2*1 = 2

----- STACK -------
|                  |
|                  |<-- toppen av stack
|  factorial(2)    |
|  factorial(3)    |
|  factorial(4)    |
--------------------
```

# Call stack for factorial(4)

```
int factorial(n) {
  if (n == 1) return 1;          //// NB!!!!!!!!!!!!
  return n*factorial(n-1);
}
```

```
factorial(3) returnerer 3*2 = 6
```

```
----- STACK -------
|                  |
|                  |
|                  |<-- toppen av stack
|  factorial(3)    |
|  factorial(4)    |
--------------------
```

# Call stack for factorial(4)

```
int factorial(n) {
  if (n == 1) return 1;          //// NB!!!!!!!!!!!!
  return n*factorial(n-1);
}
```

```
factorial(4) returnerer 4*6 = 24

----- STACK -------
|                  |
|                  |
|                  |
|                  |<-- toppen av stack
|  factorial(4)    |
--------------------
```

# Call stack for factorial(4)

```
int factorial(n) {
  if (n == 1) return 1;          //// NB!!!!!!!!!!!!
  return n*factorial(n-1);
}
```

```
factorial(4) returnerer 4*6 = 24

----- STACK -------
|                  |
|                  |
|                  |
|                  |
|                  |<-- toppen av stack
--------------------
```

Dette hadde ikke fungert hvis `factorial` hadde en fast plass i minnet til sine data. Bl. a. ville vi mistet returpeker.

# Aktivitet: Fibonacci-rekka https://bit.ly/3BivCao (løsningsforslag https://bit.ly/3l6586u)

- $F_n = F_{n-1} + F_{n-2}$
- $F_1 = F_2 = 1$
- 1 1 2 3 5 8 13 ...
- **Oppgaven:** lag en rekursiv funksjon `int fib(int n) { ... }` som regner ut det `n`-te Fibonacci-tallet. Hva er $F_{20}$?
- Spm: hvorfor er dette så ineffektivt? Hvor stor `n` fungerer dette for? Hvor mange funksjonskall gjør vi?

**"Extra credit:"** Søk i sortert array med rekursjon:
https://tinyurl.com/binarysearch2023