

# Föreläsning 1

---

Tobias Wrigstad

*Kursansvarig*

*Välkommen, nu kör vi!*



# Imperativ- och objektorienterad programmeringsmetoder

---

- Du kan ”tänka programmering” — efter PKD

Funktionell programmering

Algoritmer, datastrukturer

- Denna fortsättningskurs fokuserar på de två vanligaste paradigmen — imperativ- & OOP

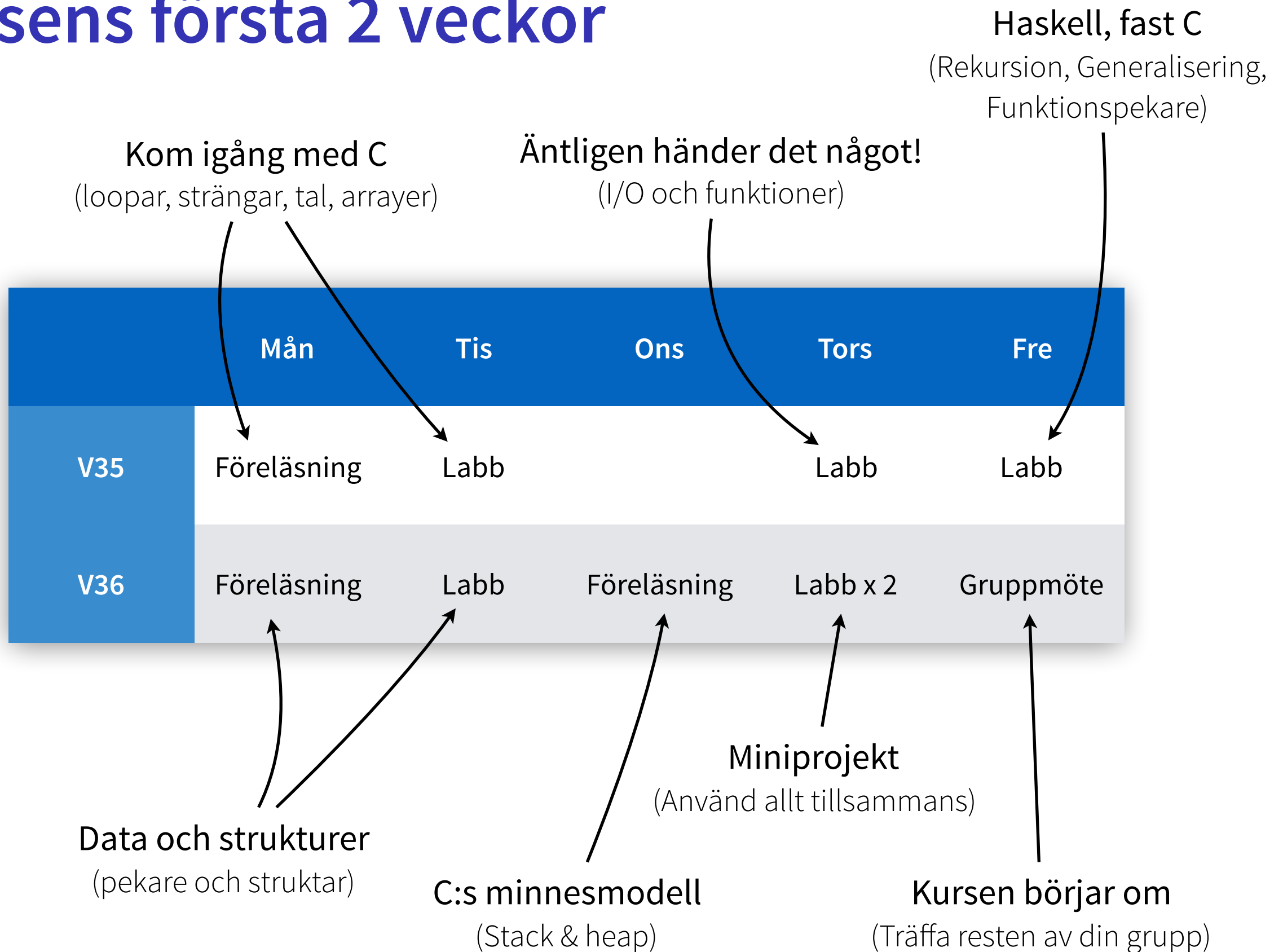
Vi skall lära oss delar av C och Java, skriva ett par 1000 rader kod (förhoppningsvis)

Denna vecka och nästa har vi 6 labbar i C i syfte att du skall komma igång

**Introduktion till kursen kommer föreläsning 4**

- IOOPM i siffror
  - ~30 föreläsningar
  - ~30 labbar
  - 4 inlämningsuppgifter
  - 1 projektuppgift
  - 30–70 olika delmål som skall redovisas var för sig, ca 2–3 i veckan
  - ~15 assistenter
  - ~10 timmar schemalagd undervisning varje vecka — minst lika mycket till krävs

# Kursens första 2 veckor



# Föreläsning 1.1

---

Tobias Wrigstad

*Kursansvarig*

*Grundläggande datatyper,  
deklaration, uttryck och satser*



**Vad är imperativ programmering?**

# C

---

- Maskinnära språk

Resurskritiska applikationer, hårdvarunära programmering, effektivitet

- Skapades ca 1969, användes för att implementera UNIX

- Språk som kan ersätta C: C++, D, Go, Java, Rust

På denna kurs använder vi C för att det inte gömmer komplexitet

# Några skillnader mellan C och Haskell

---

- C är imperativt och eager ("ivrigt"), Haskell är funktionellt och lazy
- Språken tillhör olika syntaxfamiljer
- C är manifest typat: alla variabler måste ges en explicit typ av programmeraren
- C är svagt typat: vissa typomvandlingar görs automatiskt och okontrollerade brutala typomvandlingar tillåts
- C har ingen list-typ
- I C kan man arbeta direkt med minnesadresser (pekare)
- Minneshanteringen i C måste ofta göras explicit
- Det görs vanligen ingen runtime-kontroll när C-program exekverar (vild adressering, arraygränser, odefinierade variabelvärden . . . )



```
#include<stdio.h>

int main(int argc, char *argv[])
{
    puts("Hello, world!");
    return 0;
}
```

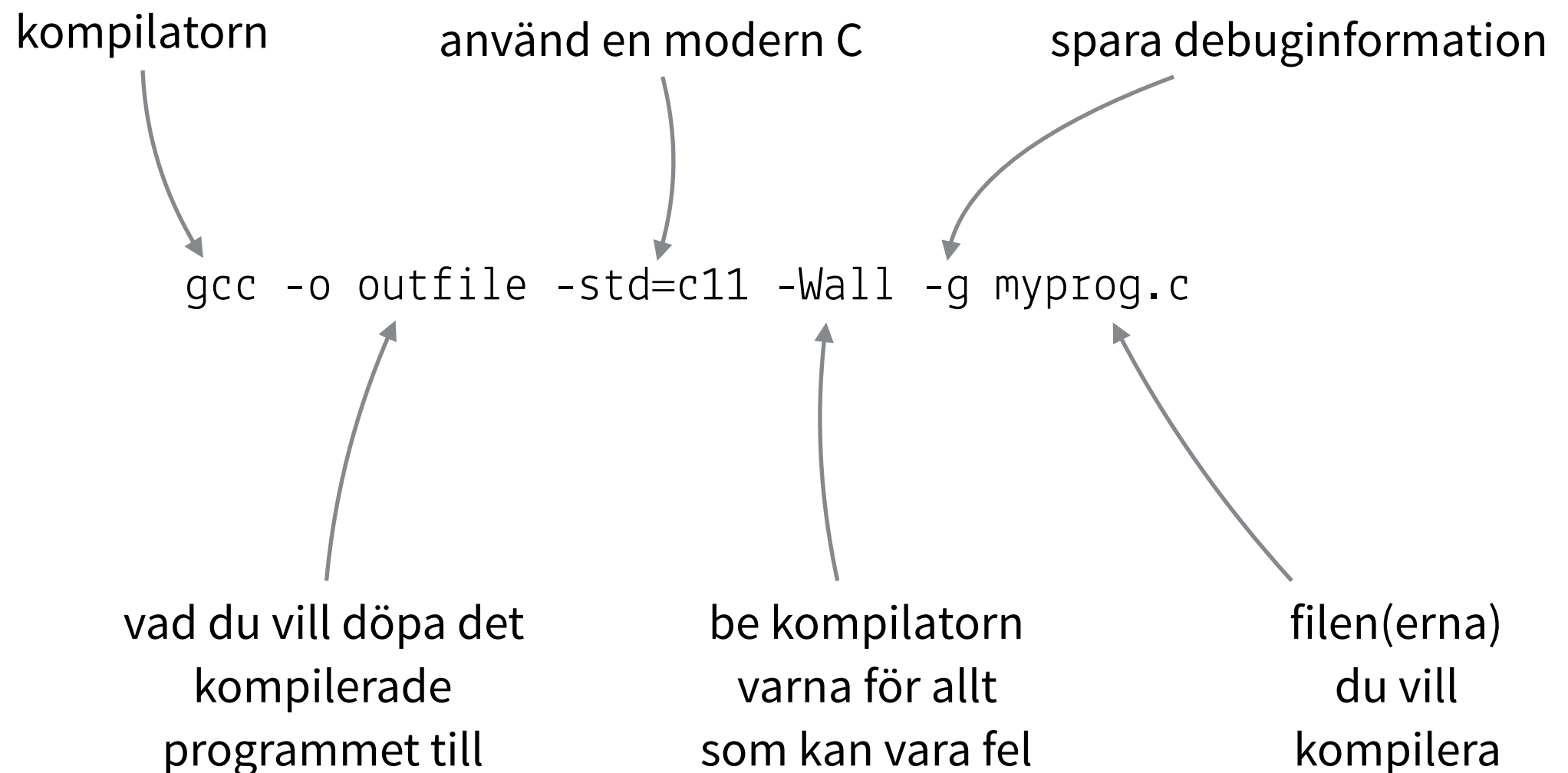
../L1/hello.c

```
$ ./hello
Hello, world!
$ _
```



# Kompilera ditt program

---



Kör ditt program: `./outfile`

# Variabeldeklaration

---

Syntax:            `typ variabelnamn;`  
                    `typ variabelnamn = expr;`

Exempel:           `int age;`  
                    `int age = 42;`

- Variabler är symboliska namn för värden

**Namnet** är extremt viktigt för det ger mening för programmeraren

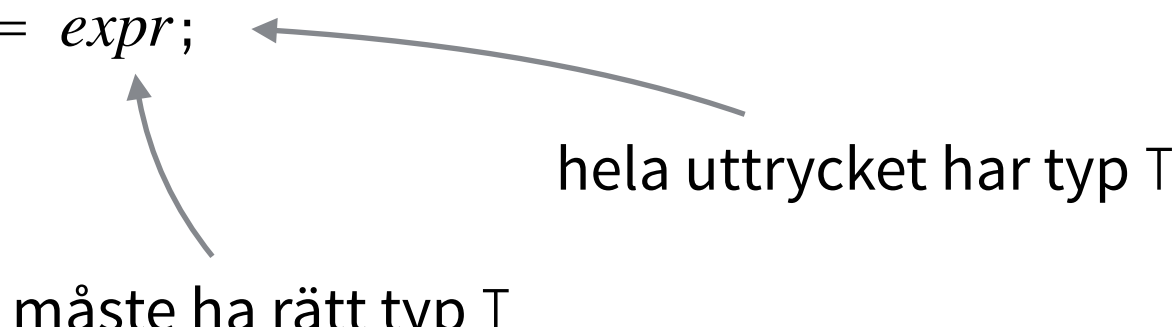
Variabers värde kan **förändras**

Oinitierade variablers värden är **odefinierade**

# Tilldelning till variabler

---

Syntax:

variabelnamn = *expr*;   
hela uttrycket har typ T  
måste ha rätt typ T

Exempel:

age = 100; // Tilldela 100 till variabeln age

age = age + 1; // Öka variabelns värde med 1

total = age = age + 1; // OK, men vansinne

# Datatyper [de vanligaste för nu]

Vanliga datatyper		
	Beskrivning	Storlek
char	Ett tecken	Minst 8 bitar
short	Litet heltal	Minst 16 bitar
int	Heltal	Minst 16 bitar
long	Stort heltal	Minst 32 bitar
float	Litet flyttal	Ospecificerat
double	Stort flyttal	Minst som float
void	Ingenting!	n/a
bool	Sedan C99	[true, false]

*Storlekarna är  
beroende av vilken  
hårdvara  
programmet är  
kompilerat på/för.*

Kräver biblioteket `stdbool.h`

[https://en.wikipedia.org/wiki/C\\_data\\_types](https://en.wikipedia.org/wiki/C_data_types)

```

#include <stdbool.h>
#include <stdio.h>

int main(void)
{
    printf("bool           %zd\n", sizeof(bool));
    printf("char           %zd\n", sizeof(char));
    printf("short          %zd\n", sizeof(short));
    printf("int            %zd\n", sizeof(int));
    printf("long           %zd\n", sizeof(long));
    printf("long long      %zd\n", sizeof(long long));
    printf("float          %zd\n", sizeof(float));
    printf("double         %zd\n", sizeof(double));
    printf("long double    %zd\n", sizeof(long double));
    return 0;
}

```

data-type-sizes.c

```

$ ./data-type-sizes
bool           1
char           1
short          2
int            4
long           8
long long      8
float          4
double         8
long double    16
$ _

```



# Operatorer

Aritmetik	
+	Addition
-	Subtraktion
*	Multiplikation
/	Division
%	Modulo
++	Inkrementera
--	Dekrementera

Relationer	
==	Likhet
!=	Olikhet
<	Strikt mindre än
<=	Mindre än
>	Strikt större än
=>	Större än

Logik	
&&	Och
	Eller
!	Negation

*Plus bitoperatorer — vi återkommer till dem senare i kursen*

# Många olika varianter av tilldelning

Kortform	Långform	Kommentar
age += 1	age = age + 1	
age -= 1	age = age - 1	
age++	tmp = age; age = age + 1; tmp	<i>Vanlig felkälla!</i>
++age	age = age + 1	
age--	tmp = age; age = age - 1; tmp	<i>Vanlig felkälla!</i>
--age	age = age - 1	
age /= 2	age = age / 2	
age *= 2	age = age * 2	



# Villkorssatser (conditionals)

---

Syntax:            **if** (*expr*) { *expr*; }  
                  **if** (*expr*) { *expr*; } **else** { *expr*; }  
                  *expr* ? *expr* : *expr*

Exempel:            **if** (age > 100) { puts("Very old"); }  
                  **if** (age % 2 == 0) { puts("Even"); } **else** { puts("Odd"); }  
                  a < b ? b : a;

- Den vanligaste formen av villkorssats returnerar inget värde
- Den något kryptiska ? : -formen har returvärde

# Läsbarhet och frihet [alla dessa är semantiskt ekvivalenta]

---

```
if (age > 100) { puts("Very old"); }
```

```
if (age > 100)
{
    puts("Very old");
}
```

```
if (age > 100) {
    puts("Very old");
}
```

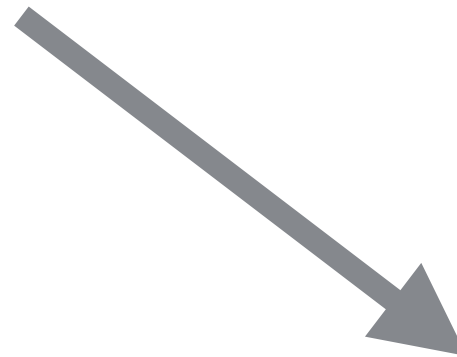
```
if (age > 100) puts("Very old");
```

```
if (age > 100)
    puts("Very old");
```

# Läsbarhet: Apples #gotofail SSL bug [1/2]

```
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
    goto fail;
goto fail;
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
    goto fail;
```

*Indenteringen ljuger!*



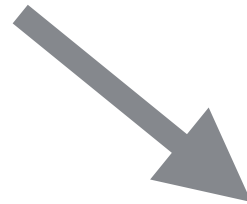
*Indenteringen lyfter fram felet!*

```
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
    goto fail;
goto fail;
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
    goto fail;
```

# Läsbarhet: Apples #gotofail SSL bug [2/2]

```
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
    goto fail;
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
    goto fail;
goto fail;
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
    goto fail;
```

*Inga block — fail!*



*Block — inget fail!*

```
if ((err = SSLHashSHA1.update(&hashCtx, &serverRandom)) != 0)
{
    goto fail;
}
if ((err = SSLHashSHA1.update(&hashCtx, &signedParams)) != 0)
{
    goto fail;
    goto fail;
}
if ((err = SSLHashSHA1.final(&hashCtx, &hashOut)) != 0)
{
    goto fail;
}
```

# Switchsatser

---

Syntax:           **switch** (*expr*)  
                  {  
                  **case** *literal* *body*; **break**;  
                  **default**: *body*;  
                  }

Exempel:           **switch** (n)  
                  {  
                  **case** 0:   result = 0; **break**;  
                  **case** 1:   result = 1; **break**;  
                  **default**: result = fib(n - 1) + fib(n - 2);  
                  }

- Vanlig felkälla — bevisat dålig design

# Exempel på en trasig switchsats

---

```
switch (n)
{
  case 0:  result = 0;
  case 1:  result = 1;
  default: result = fib(n - 1) + fib(n - 2);
}
```

- Vad händer om  $n == 1$ ?

# Iteration med loopar: while

Syntax:

```
while (cond) { body }
```

```
while (cond) expr;
```

*Om sant, gå ett  
"till varv" i loopen*

*"Loop-kroppen" — det  
som körs varje "varv"*

Exempel:

```
int n_fakultet = 1;
```

```
int n = 6;
```

```
while (n >= 1)
```

```
{
```

```
    n_fakultet *= n;
```

```
    n = n - 1;
```

```
}
```

```
printf("%d! = %d\n", n, n_fakultet);
```

# Loopar är bekväma och nödvändiga

```
// n == 6  
while (n)  
    n_fakultet *= n--;
```

← *Vad du skrev*

```
n_fakultet *= n--;  
n_fakultet *= n--;  
n_fakultet *= n--;  
n_fakultet *= n--;  
n_fakultet *= n--;  
n_fakultet *= n--;
```

*Vad kompilatorn gjorde (unrolling)  
(bara möjligt om  $n == 6$  går att avgöra)*

←



# Loopar är bekväma och nödvändiga

```
// n == 6  
while (n)  
    n_fakultet *= n--;
```

← *Vad du skrev*

```
n_fakultet *= 6;  
n_fakultet *= 5;  
n_fakultet *= 4;  
n_fakultet *= 3;  
n_fakultet *= 2;  
n_fakultet *= 1;
```

*Vad kompilatorn gjorde (unrolling)  
(bara möjligt om  $n == 6$  går att avgöra)*

←

# Läsbarhet [identiska satser enligt kompilatorn]

---

```
while (n >= 1)
{
    n_fakultet *= n;
    n = n - 1;
}
```

```
while (n)
{
    n_fakultet *= n--;
}
```

```
while (n) n_fakultet *= n--;
```

```
while (n)
    n_fakultet *= n--;
```

# Kort utvikning: do-while

---

```
do
{
    n_fakultet *= n;
}
while (n--);
```

# Iteration med loopar: for

Syntax:

```
for (init; pre; post) { body }
```

```
for (init; pre; post) expr;
```

*Deklarera och initiera  
loopvariabler*

*Om sant, gå ett  
"till varv" i loopen*

*Utförs alltid sist  
i varje varv*

Exempel:

```
int n_fakultet = 1;  
for (int i = 1; i <= n; i = i + 1)  
{  
    n_fakultet *= i;  
}
```

```
printf("%d! = %d\n", n, n_fakultet);
```

# Main – där alla C-program börjar

---

*Antalet kommandorads-  
argument*

*De faktiska  
kommandorads-  
argumenten*

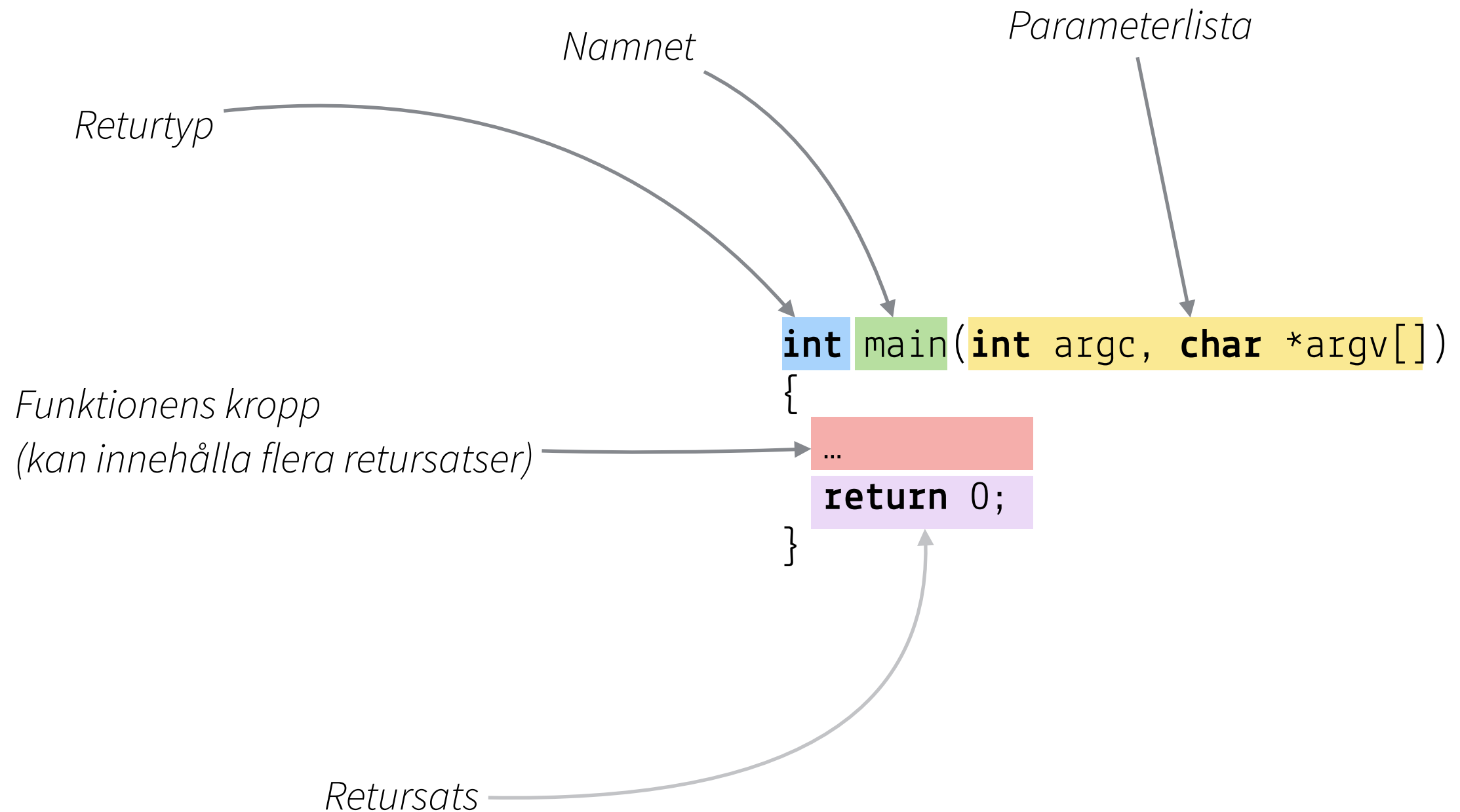
```
int main(void)
{
    ...
    return 0;
}
```

OK

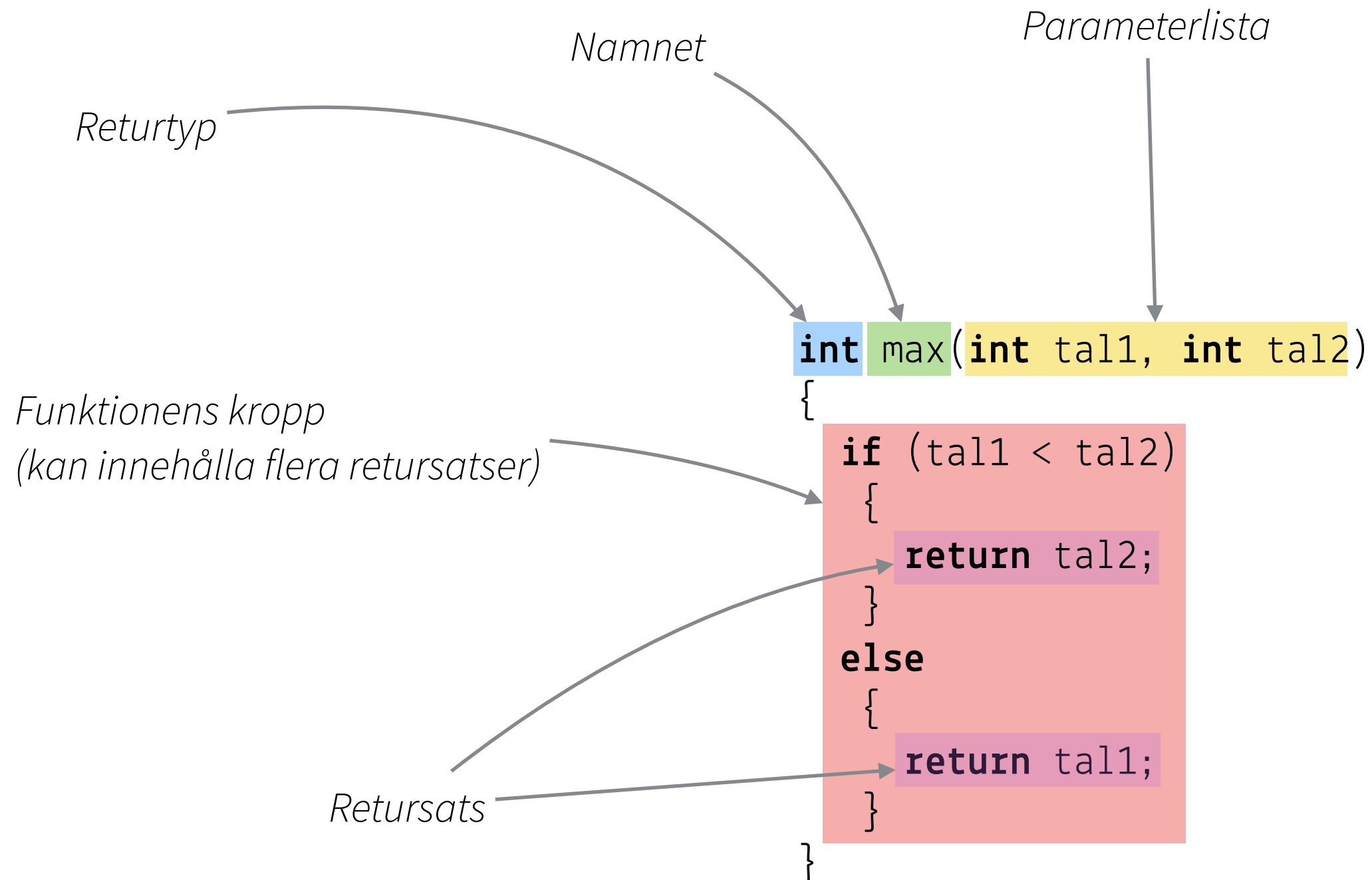
```
int main(int argc, char *argv[])
{
    ...
    return 0;
}
```

Bättre

# Funktionens anatomi



# Deklarera egna funktioner



# Deklarera egna funktioner

---

*// Exempel på anrop*

```
int a = max(512, 1024);
```

```
int max(int tal1, int tal2)
{
    if (tal1 < tal2)
    {
        return tal2;
    }
    else
    {
        return tal1;
    }
}
```



# Inkludera funktioner från andra bibliotek

---

**#include** <filnamn.h> ← Inkludera från standardbibliotek

**#include** "filnamn.h" ← Inkludera från ditt eget program

*Plus extra länkning i kompileringssteg. Vi återkommer till det senare.*

# Exempel på olika funktioner

---

Funktion	Kommentar
<b>void</b> puts( <b>char</b> *)	Skriv ut en sträng på skärmen
<b>int</b> atoi( <b>char</b> *)	Konvertera en sträng till ett heltal (int)
<b>long</b> atol( <b>char</b> *)	Konvertera en sträng till ett heltal (long)
<b>int</b> getchar()	Läs in ett tecken från tangentbordet
FILE *fopen( <b>char</b> *, <b>char</b> *)	Öppna en fil

*Läs mer om funktionerna med hjälp av man-kommandot*

# Läsbarhet [identiska funktioner enligt kompilatorn]

---

```
int max(int tal1, int tal2)
{
    if (tal1 < tal2)
    {
        return tal2;
    }
    else
    {
        return tal1;
    }
}
```

```
int max(int tal1, int tal2)
{
    return (tal1 < tal2) ? tal2 : tal1;
}
```

```
int max(int tal1, int tal2)
{
    if (tal1 < tal2) return tal2;
    return tal1;
}
```

# Returns kontrollflöde [vanlig felkälla]

---

```
int max(int tal1, int tal2)
{
    if (tal1 < tal2)
    {
        return tal2;
    }
    else
    {
        return tal1;
    }

    puts("Jag skrivs aldrig ut!");
}
```

# Program som skriver ut kommandoradsargument

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    printf("%d kommandoradsargument\n", argc);
    for (int i = 0; i < argc; ++i)
    {
        printf("Argument %d = %s\n", i, argv[i]);
    }
    return 0;
}
```

cl-args.c



# Komplett exempel för n-fakultet

```
#include <stdlib.h>
#include <stdio.h>

int main(int argc, char *argv[])
{
    int nfak = 1;
    int n = atoi(argv[1]);
    for (int i = 1; i <= n; ++i)
    {
        nfak *= i;
    }
    printf("%d! = %d\n", n, nfak);
    return 0;
}
```

nfak.c



# Arrayer

---

`T name[size];` // deklarerar en array av *size* element av typen T

```
int salaries[500];
```

```
long sum = 0;
```

```
for (int i = 0; i < 500; ++i)
```

```
{
```

```
    sum += salaries[i];
```

```
}
```

*läs det i:te elementet i arrayen*



- Arrayer har en fix storlek – kan inte ändras
- Arrayer indexeras  $[0, size)$  – första elementet har index 0, sista  $size-1$

Skriv: `myarr[17] = 42;`      Läs: `myarr[x]`

- Arrayerna har inget metadata, och C gör ingen indexkontroll

# Ingen indexkontroll

---

```
int salaries[500];  
long sum = 0;  
  
for (int i = 0; i <= 500; ++i)  
{  
    sum += salaries[i];  
}
```

*Vad blir resultatet av detta program när det körs?*



# Ingen indexkontroll

---

```
int salaries[500];  
long sum = 0;  
  
for (int i = 0; i <= 500; ++i)  
{  
    sum += salaries[i];  
}
```

**UNDEFINED  
BEHAVIOUR**

*Vad blir resultatet av detta program när det körs?*

# Kommentarer

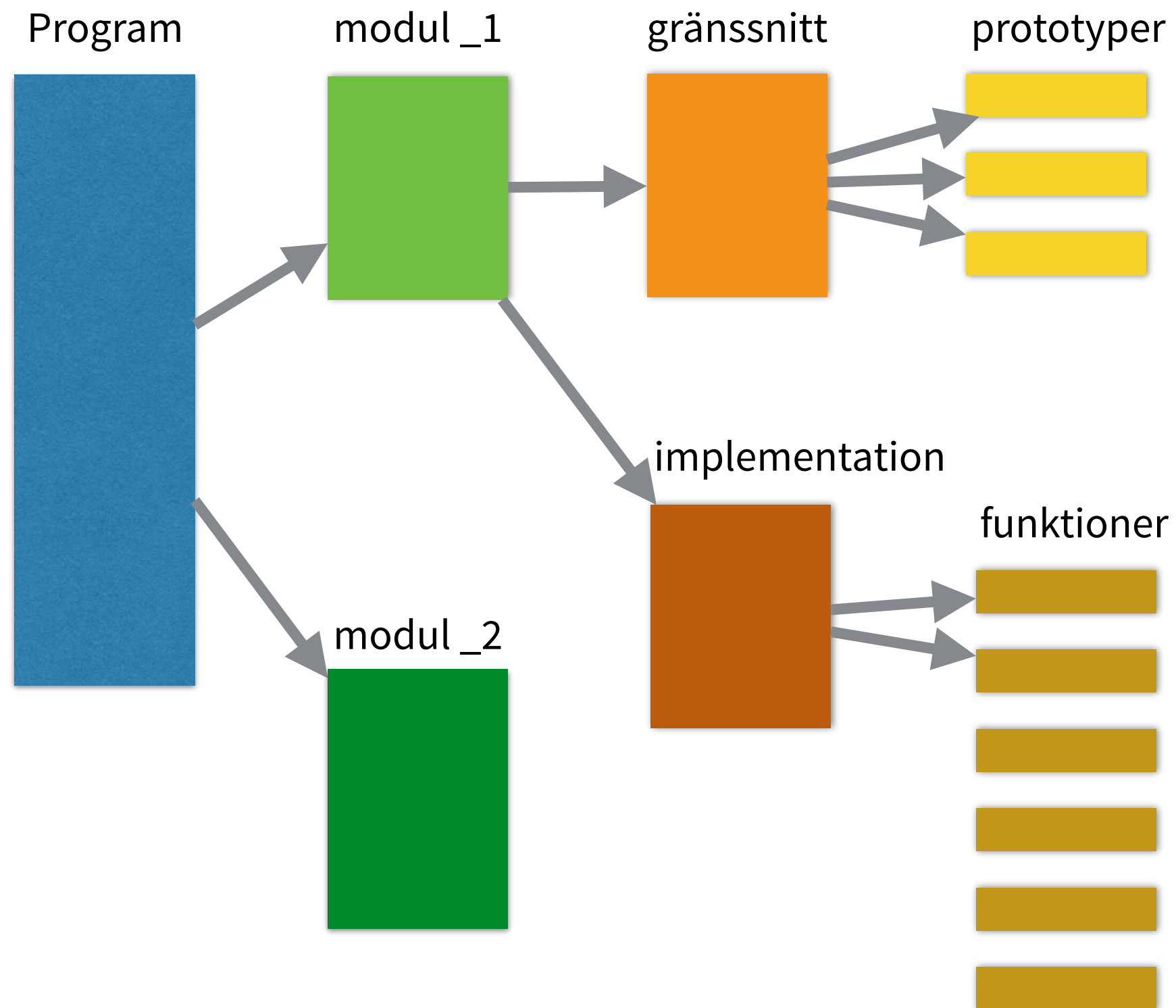
```
// Startar kommentar som gäller till radens slut
```

```
/* Startar kommentarblock som gäller ända till */
```

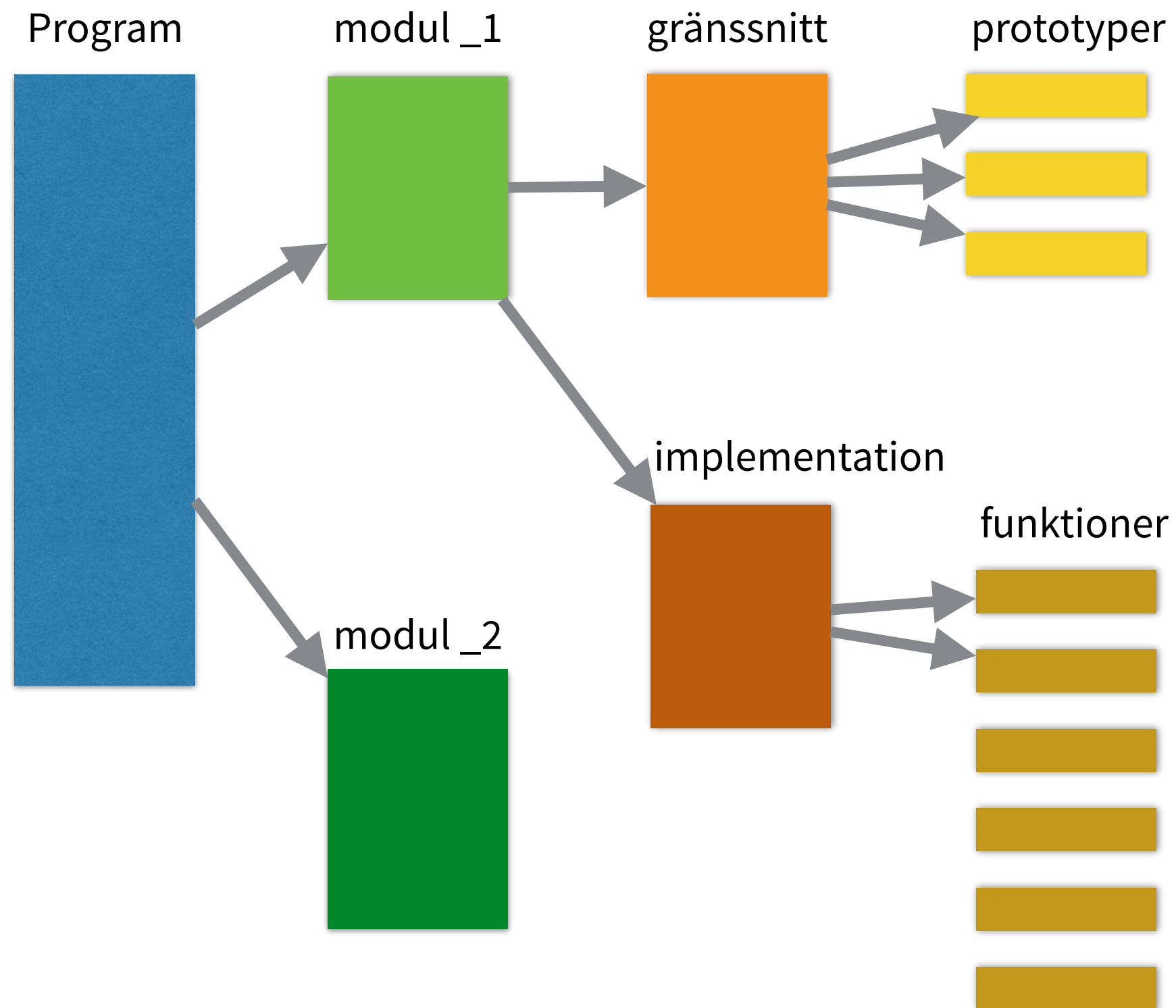
Kommentarer är mest nödvändiga för att förklara *varför*.

Om du känner att du behöver kommentera en bit kod för att den skall gå att förstå är det 99% chans att koden borde skrivas om istället för kommenteras.

# Ett programs anatomi



# Ett programs anatomi



# Funktionsabstraktionen

prototyper

→ **void** add\_item\_to\_store(...);

→ **bool** in\_stock(...);

→ **int** get\_price(...);

funktioner

→ *kod*

→ *kod*

*kod*

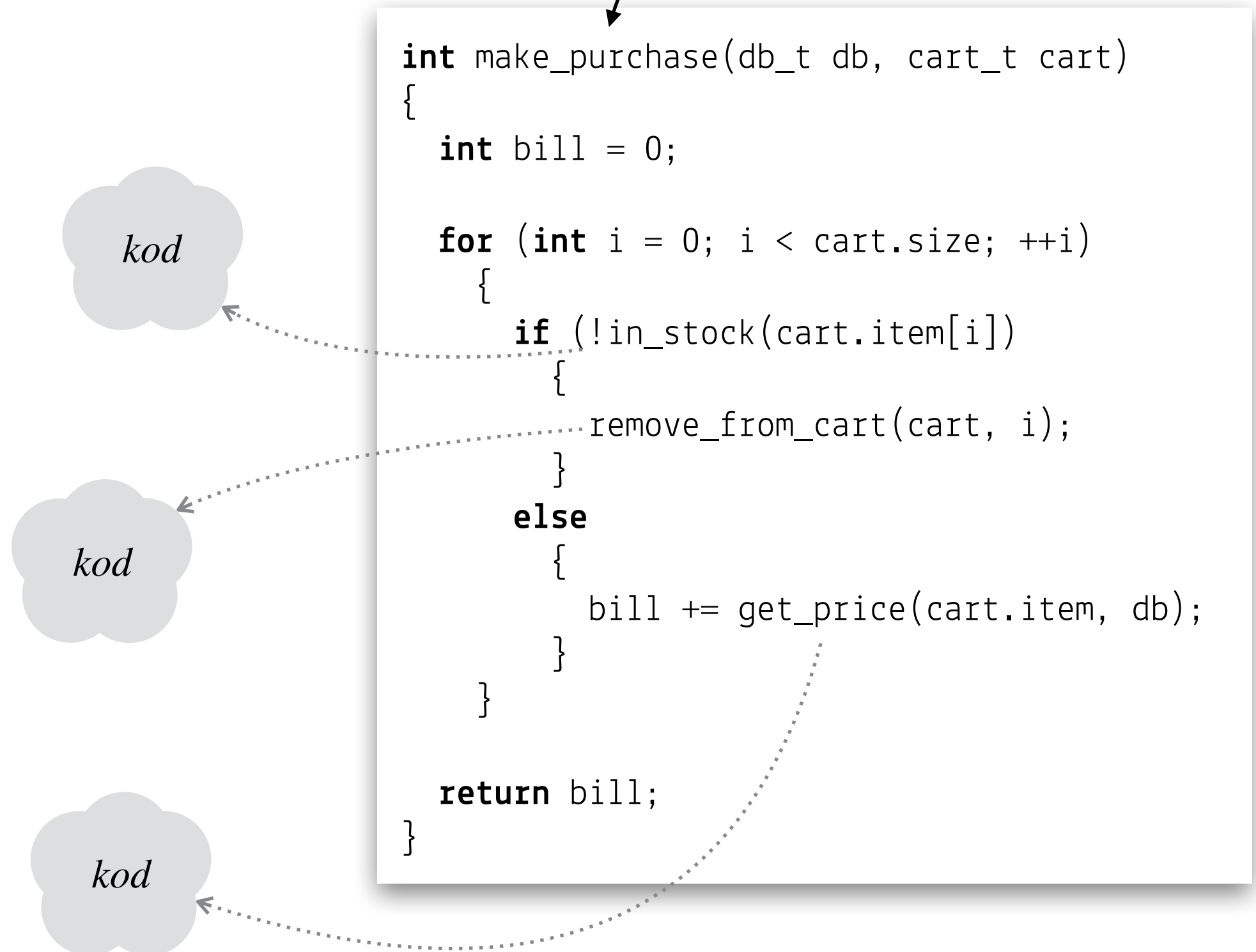
*kod*

*kod*



# Funktionsabstraktionen

*Bygga abstraktioner av abstraktioner!*



# Föreläsning 1.9

---

*Gås endast igenom kursivt på föreläsningen och i mån av tid  
— dock bra att ha tillgängligt inför lab 2 & 3*

*Se även **extramaterialet** i föreläsningsanteckningarna.*

*Introduktion till standard-I/O och  
grundläggande stränghantering*



# Hur var det i Haskell?

---

- Char är en egen typ

`'a' :: Char`

`'a' + 2` — *kompilerar ej*



# Hur var det i Haskell?

---

- Char är en egen typ

`'a' :: Char`

`'a' + 2` — *kompilerar ej*

- `type String = [Char]`

`"Hello" == ['H', 'e', 'l', 'l', 'o']`

`'F' : "oo" ++ "Bar!" == "FooBar!"`

# Hur var det i Haskell?

---

- Char är en egen typ

`'a' :: Char`

`'a' + 2` — *kompilerar ej*

- `type String = [Char]`

`"Hello" == ['H', 'e', 'l', 'l', 'o']`

`'F' : "oo" ++ "Bar!" == "FooBar!"`

- Kan du listor kan du strängar! (mer eller mindre)

# Tecken i C

---

- **char** — ett heltal (på *normalt* en byte)
- Ett tecken är en **char** med motsvarande ASCII-värde

```
char c1 = 'a';
```

```
char c2 = 97;      == c1
```

```
char c3 = 'a' + 2; == 'c'
```

```
char c4 = '2' + '2'; == 50 + 50
```

# Tecken i C

---

- **char** — ett heltal (på *normalt* en byte)
- Ett tecken är en **char** med motsvarande ASCII-värde

```
char c1 = 'a';
```

```
char c2 = 97;      == c1
```

```
char c3 = 'a' + 2; == 'c'
```

```
char c4 = '2' + '2'; == 100
```

# Tecken i C

---

- **char** — ett heltal (på *normalt* en byte)
- Ett tecken är en **char** med motsvarande ASCII-värde

```
char c1 = 'a';
```

```
char c2 = 97;      == c1
```

```
char c3 = 'a' + 2; == 'c'
```

```
char c4 = '2' + '2'; == 'd'
```

# Tecken i C

---

- **char** — ett heltal (på *normalt* en byte)
- Ett tecken är en **char** med motsvarande ASCII-värde

```
char c1 = 'a';
```

```
char c2 = 97;      == c1
```

```
char c3 = 'a' + 2; == 'c'
```

```
char c4 = '2' + '2'; == 'd'
```

- Obs! 'a' och 97 är alltså ekvivalenta

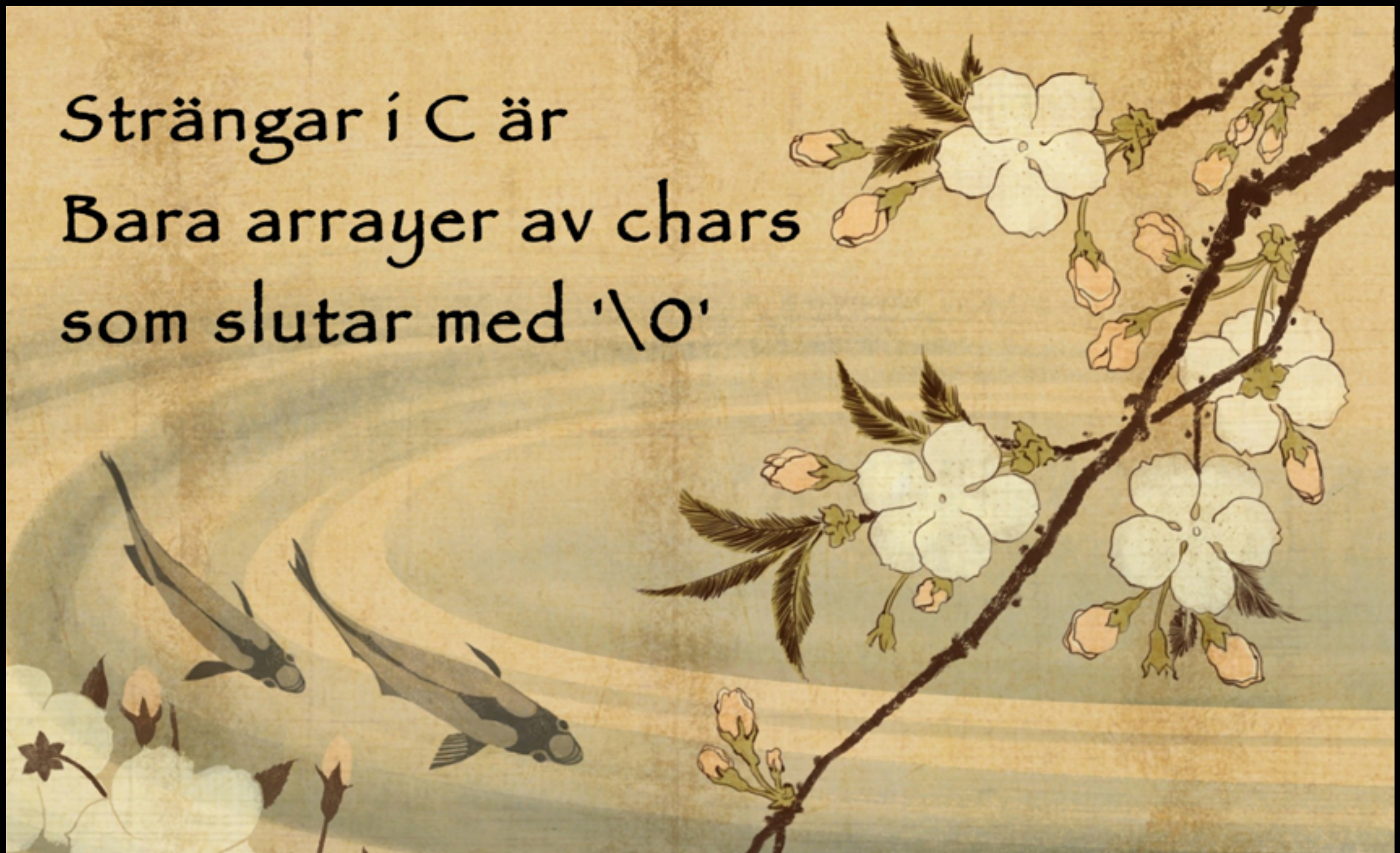
'a' är *syntaktiskt socker* för 97

# Tecken i C

---

- Utdelat exempel: `ascii.c`

Strängar i C är  
Bara arrayer av chars  
som slutar med '\0'





# Strängar i C

---

- C har ingen strängtyp!

`char*` — en pekare till en array av chars

`char[]` — en array av chars

I stora stycken är typerna ovan ekvivalenta

# Strängar i C

---

- C har ingen strängtyp!

`char*` — en pekare till en array av chars

`char[]` — en array av chars

I stora stycken är typerna ovan ekvivalenta

- Kan du listor kan du strängar! (mer eller mindre)

`char s[] = "Hello"` är **precis** samma sak som

`char s[] = {'H', 'e', 'l', 'l', 'o', '\0'}` vilket är **precis** samma sak som

`char s[] = {72, 101, 108, 108, 111, 0}`

# Strängar i C

---

- C har ingen strängtyp!

`char*` — en pekare till en array av chars

`char[]` — en array av chars

I stora stycken är typerna ovan ekvivalenta

- Kan du listor kan du strängar! (mer eller mindre)

`char s[] = "Hello"` är **precis** samma sak som

`char s[] = {'H', 'e', 'l', 'l', 'o', '\0'}` vilket är **precis** samma sak som

`char s[] = {72, 101, 108, 108, 111, 0}`

s =	'H'	'e'	'l'	'l'	'o'	'\0'
-----	-----	-----	-----	-----	-----	------

# Strängar i C

---

- C har ingen strängtyp!

`char*` — en pekare till en array av chars

`char[]` — en array av chars

I stora stycken är typerna ovan ekvivalenta

- Kan du listor kan du strängar! (mer eller mindre)

`char s[] = "Hello"` är **precis** samma sak som

`char s[] = {'H', 'e', 'l', 'l', 'o', '\0'}` vilket är **precis** samma sak som

`char s[] = {72, 101, 108, 108, 111, 0}`

s = 

'H'	'e'	'l'	'l'	'o'	'\0'
-----	-----	-----	-----	-----	------

*Hello är en array av sex chars!*

# Vanliga misstag 1

- Strängjämförelse med `==` avser *identitet*, inte *ekvivalens*

```
#include <stdio.h>

int main (void)
{
    char password[] = "abc123";
    char entered[128];

    puts("Please enter the secret code:");
    scanf("%s", entered); // read input

    if (entered == password)
    {
        puts("You are logged in!");
    }
    else
    {
        puts("Incorrect password!");
    }

    return 0;
}
```

# Vanliga misstag 1: Strängjämförelse

---

- Utdelat exempel: `password.c`

```
#include <stdio.h>

int main (void)
{
    char password[] = "abc123";
    char *entered;      // kommer att "peka ut" inläst data
    size_t entered_size; // längden på entered

    puts("Please enter the secret code:");
    getline(&entered, &entered_size, stdin); // read input

    if (entered == password)
    {
        puts("You are logged in!");
    }
    else
    {
        puts("Incorrect password!");
    }

    return 0;
}
```

password-bad.c



```
#include <stdio.h>

int main (void)
{
    char password[] = "abc123";
    char *entered;      // kommer att "peka ut" inläst data
    size_t entered_size; // längden på entered

    puts("Please enter the secret code:");
    getline(&entered, &entered_size, stdin); // read input

    if (strncmp(entered, password, entered_size) == 0)
    {
        puts("You are logged in!");
    }
    else
    {
        puts("Incorrect password!");
    }

    return 0;
}
```

password-good.c





# Vanliga misstag 2

- Aliasering!

*Inläsning med `getline()`  
undviker detta!*

```
#include <stdio.h>

int main (void)
{
    char buffer [128];
    char *first;
    char *last;

    puts("What is your first name?");
    scanf("%s", buffer);
    first = buffer;

    puts("What is your last name?");
    scanf("%s", buffer);
    last = buffer;

    printf("Hello %s %s!\n", first, last);
    return 0;
}
```

# Vanliga misstag 2: Aliasering

---

- Se exempel: `greeting.c`

# Vanliga misstag 3

- ...som slutar med '\0'!

```
#include <stdio.h>

void copy(char to[], char from[], int len)
{
    while(len--)
        to[len] = from[len];
}

int main (void)
{
    char *s = "Hello";
    char t[5];

    copy(t, s, 5);

    puts(t);

    return 0;
}
```

# Vanliga misstag 3: ...som slutar med '\0'!

---

- Se exempel: `length.c`

# Standardbiblioteket `string.h`

- Inkluderas med `#include <string.h>`

Funktion	Beskrivning
<code>strlen(s)</code>	längden av <code>s</code> (utan <code>'\0'</code> -tecknet!)
<code>strncpy(s, t, n)</code>	kopiera <code>t</code> till <code>s</code> (upp till <code>n</code> tecken)
<code>strncmp(s, t, n)</code>	jämför <code>s</code> och <code>t</code> (upp till <code>n</code> tecken)
<code>== 0</code>	<code>s</code> och <code>t</code> är samma sträng
<code>&gt; 0</code>	<code>s</code> kommer efter <code>t</code> i bokstavsordning
<code>&lt; 0</code>	<code>s</code> kommer före <code>t</code> i bokstavsordning
<code>strncat(s, t, n)</code>	Lägg <code>t</code> följt av <code>s</code> i <code>t</code>

# Funktioner i `stdlib.h` (togs även upp tidigare)

---

- Inkluderas med `#include <stdlib.h>`

`atoi(s)` — konvertera strängrepresentation av tal till motsvarande heltal

`atol(s)` — med `long` istf `int` som returtyp

Exempel:

```
atoi("123abc") == 123 (som int)
```

# OBS!

---

- Alla funktioner är optimistiska

Förutsätter från att det finns tillräckligt minne i datat som används

Förutsätter att strängar är '`\0`'-terminerade korrekt

- Använd alltid `strncpy` (`strncpy`, ...) över `strcpy` (`strcpy`, ...) etc.

# OBS!

---

- Alla funktioner är optimistiska

Förutsätter från att det finns tillräckligt minne i datat som används

Förutsätter att strängar är `'\0'`-terminerade korrekt

- Använd alltid `strncpy` (`strncpy`, ...) över `strcpy` (`strcpy`, ...) etc.
- Använd alltid funktioner från standardbiblioteken (som `string.h`) över funktioner du skriver själv



# OBS!

---

- Alla funktioner är optimistiska

Förutsätter från att det finns tillräckligt minne i datat som används

Förutsätter att strängar är `'\0'`-terminerade korrekt

- Använd alltid `strncpy` (`strncpy`, ...) över `strcpy` (`strcpy`, ...) etc.
- Använd alltid funktioner från standardbiblioteken (som `string.h`) över funktioner du skriver själv

Även om det är en bra övning att ha implementerat motsvarande funktioner själv någon gång!

# Standardbiblioteket `ctype.h`

- Inkluderas med `#include <ctype.h>`

Funktion	Beskrivning
<code>isalpha(c)</code>	Är <code>c</code> en bokstav?
<code>isdigit(c)</code>	Är <code>c</code> en siffra?
<code>islower(c)</code>	Är <code>c</code> en gemen? (liten bokstav)
<code>digittoint(c)</code>	Konvertera från '2' till 2 (som int)
<code>toupper(c)</code>	Från gemen till versal ('a' till 'A')