Imperativ och objektorienterad programmeringsmetodik

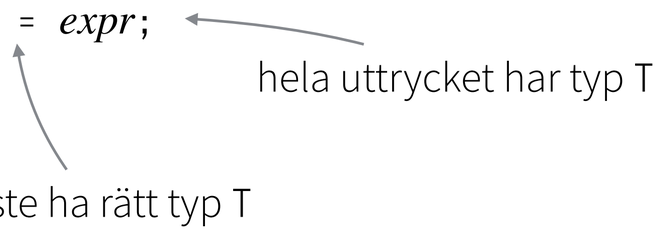
Föreläsning 2 av många

Tobias Wrigstad

Recap: vad har ni sett hittills på labbarna?

Tilldelning till variabler

Syntax: `variabelnamn = expr;`



Exempel:

```
age = 100;                  // Tilldela 100 till variabeln age

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

total = age = age + 1; // OK i C, men vansinne
```

Grundläggande datatyper i C

Moderna heltalstyper

int8_t, int16_t, int32_t, int64_t

uint8_t, …

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]

Kräver biblioteket `stdbool.h`

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

https://en.wikipedia.org/wiki/C_data_types

Grundläggande 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

Main, där alla C-program börjar

```c
int main(void)
{
  …
  return 0;
}
```

OK

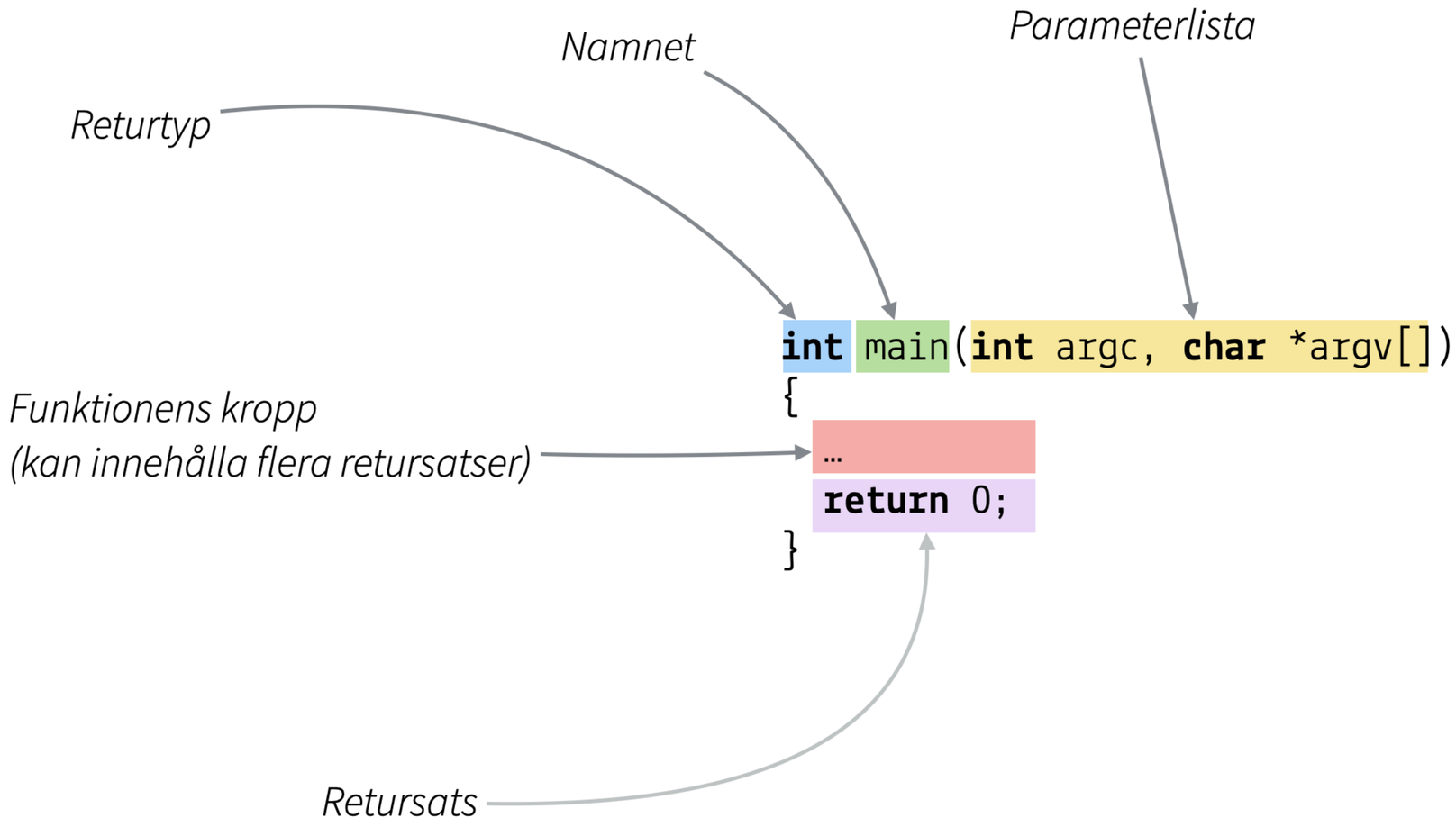
Antalet kommandorads-argument

De faktiska kommandorads-argumenten

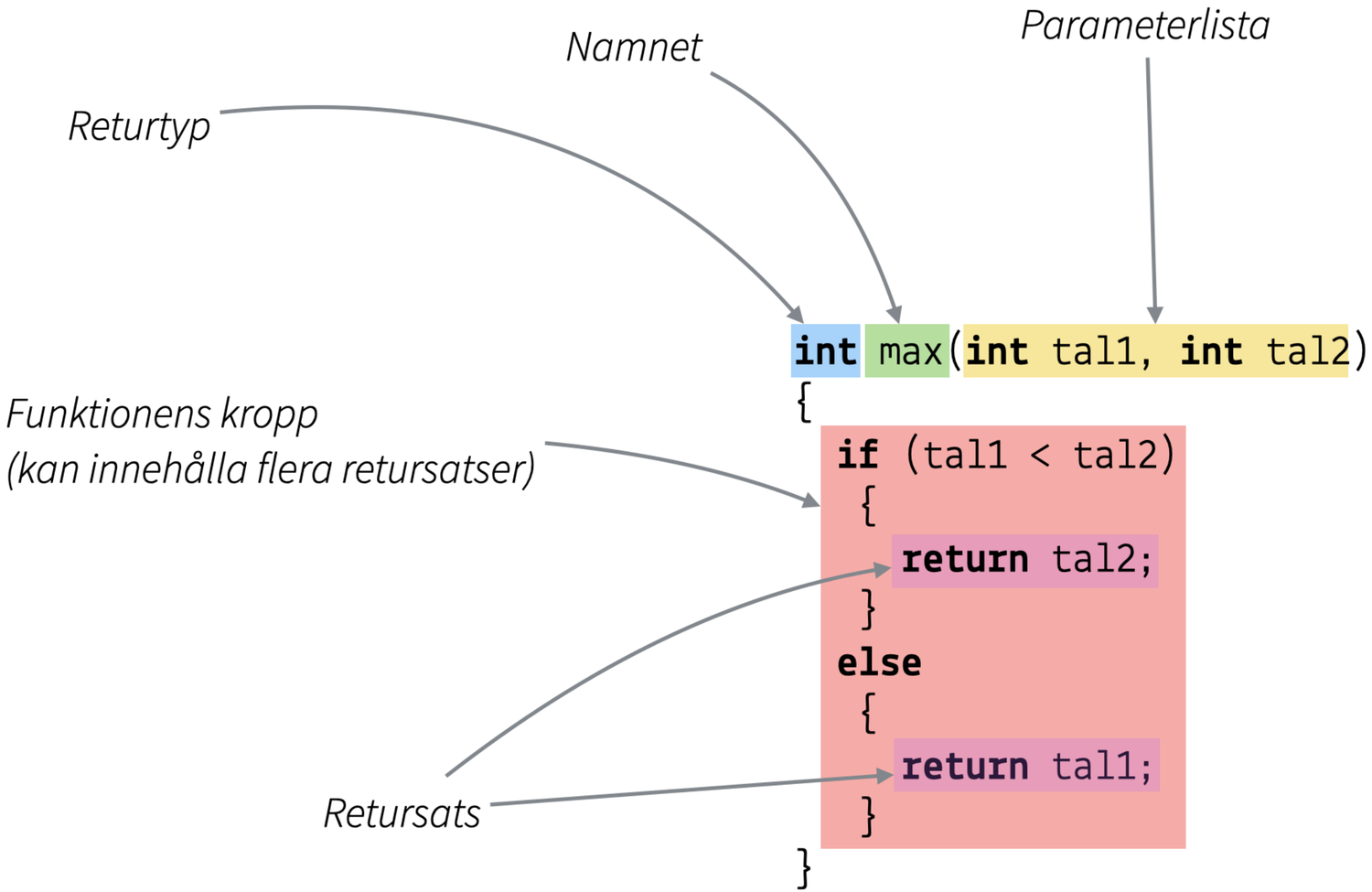
```c
int main(int argc, char *argv[])
{
  …
  return 0;
}
```

Bättre

En funktions anatomi



Att deklarera egna funktioner



Normal-order vs. Applicative order

Call by name

Evaluera vänstras förekomsten av en funktion (Haskell)

```
sum(3 + 7)

(3 + 7) * (3 + 7)

(10) * (3 + 7)

(10) * (10)

100
```

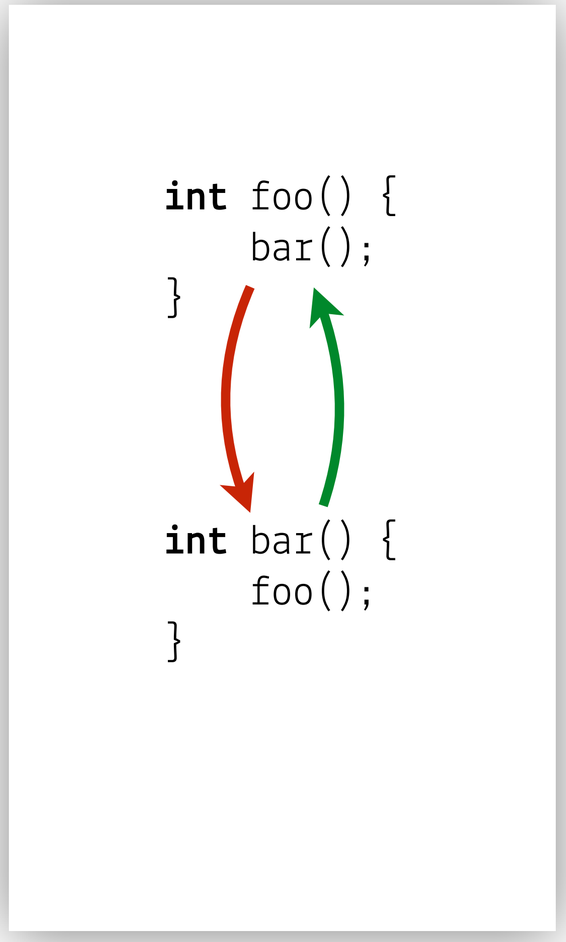
Call by value

Evaluera innersta förekomsten av en funktion (C, Java)

```
sum(3 + 7)

sum(10)

10 * 10

100
```

```
int square(int term) {
    return term * term;
}
```

C-kompilatorn är en one-pass compiler



Programmet läses uppifrån och ned

Kompilatorn känner bara till deklarationer den redan har sett

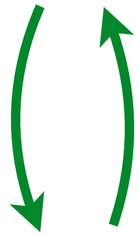
En senare funktion kan anropa en tidigare, men inte tvärtom

Lösning: ändra ordningsföljd

Problem: cirkulära beroenden

C-kompilatorn är en one-pass compiler

```
int foo();
int bar();



int foo() {
    bar();
}

int bar() {
    foo();
}
```



Programmet läses uppifrån och ned

- Kompilatorn känner bara till deklarationer den redan har sett
- En senare funktion kan anropa en tidigare, men inte tvärtom
  - Lösning: ändra ordningsföljd
  - Problem: cirkulära beroenden

Cirkulära beroenden löses med funktionsprototyper

- Forward-deklarerar förekomsten av en funktion genom att ange dess signatur
- Tjänar också till dokumentation och en plats för dokumentation

Villkorssatser

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

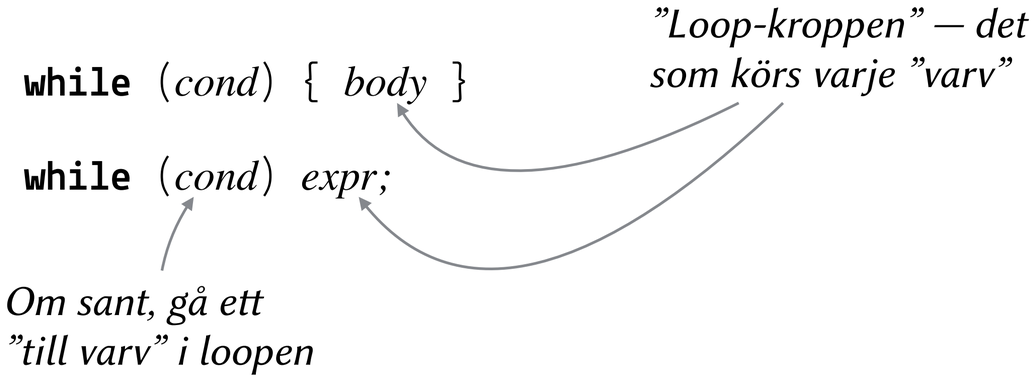
Finns även en switch-sats som är relaterad

Loopar

Syntax:

while (*cond*) { *body* }

while (*cond*) *expr;*



Exempel:

```
int n_fakultet = 1;
int n = 6;

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

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

C’s inställning till sanningsvärden

C hade länge inte någon typ för booleska värden

Alla värden som är 0 eller kan konverteras till 0 (t.ex. 0.0 eller pekaren NULL som vi träffar nästa vecka) tolkas som falska

Alla andra värden tolkas som sanna

Ger upphov till satser som till en början kan upplevas o-intuitiva

```
while (n -= 1) {
    printf(”%d\n”, n);
}

while ((n -= 1)  != 0) { … }

while (n > 1) {
    n -= 1;
    printf(…);
}
```

Loopar

```
// 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)

Se även for-loop och do while-loop

Loopar

```
// 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)

Se även for-loop och do while-loop

Läsbarhet är viktigt!

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

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

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

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

For-loopar i C

Syntax:

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

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

Observera att det inte finns något som for-loopar kan som inte while-loopar kan göra.

Exempel:

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

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

Läsbarhet och frihet

```
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”);
```

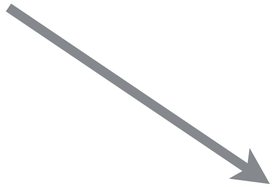
För kompilatorn är alla dessa satser ekvivalenta!

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;
```

https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-1266

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;
```

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;
}
```

https://web.nvd.nist.gov/view/vuln/detail?vulnId=CVE-2014-1266

Mentala modeller för hur program exekverar

De flesta program ni skrev under PKD kan beskrivas utmärkt med substitutionsmodellen

Ett program exekverar genom att dess text *skrivs om* tills dess att den inte går att *reducera* ytterligare

Tilldelning och funktionsanrop hanteras genom substitution

Exempel:

```
int fakultet(int n) {
    return n == 1
        ? 1
        : n * fakultet(n - 1);
}

fakultet(5);
```

Demo substitutionsmodellen

Mentala modeller för hur program exekverar

De flesta program ni skriver under IOOPM kan inte beskrivas med substitutionsmodellen

Data med föränderligt tillstånd ger att samma kodrad / uttryck kan ha olika effekt / returnera olika resultat

Tillstånd leder till tankar om ”tid” och vad som är den ”aktuella satsen”

Exempel:

```
int account = 1000;

int withdraw(int amount) {
    return account -= 100;
}

withdraw(100); // returnerar 900
withdraw(100); // returnerar 800
```

```
int fakultet(int n) {
    int sum = 1;
    for (int i = 1; i <= n; i += 1) {
        sum = sum * i;
    }
    return sum;
}
```

Utkast till modell: arbetsminne — tallriksmodellen

Minnet är organiserat som en stapel av tallrikar

Varje funktion lagrar sitt temporära data — dess parametrar och lokala variabler — på en ”tallrik”

Varje gång en funktion anropas lägger vi en ny tallrik högst upp på stapeln och lägger en kopia av dess argument därpå

När en funktion returnerar tar vi bort dess tallrik ur stapeln

En funktion kan bara komma åt värden på dess egen tallrik

Enda sättet funktioner kan dela data är genom parameteröverföring (uppåt i stapeln) och returvärden (nedåt i stapeln)

Stapeln har en maxhöjd — för många tallrikar staplade på varandra leder till en krasch

Endast funktionen på den översta tallriken får exekvera — programmet är klart när stapeln är tom

Utkast till modell: långtidsminne — rutat papper-modellen

Långtidsminnet är ett rutat pappersark där varje ”värde” i programmet utgör en eller flera rutor

- Varje ruta har en unik ”adress” i form av en (x,y)-koordinat, t.ex. (0,0) för rutan i övre vänstra hörnet
- Globala variabler ligger inte på någon tallrik, och motsvarar koordinater på papperet
- En ruta är antingen använd eller ledig
- Vid varje givet tillfälle kan en ruta användas till max ett värde

Man kan när som helst komma åt vilken ruta som helst på arket, använd eller ledig

Det rutade papprets storlek är ändligt

Demo tallriksmodellen

Stack och heap

Många programspråk exekverar i en ”miljö” som liknar tallrik- och pappersmodellen

Vi kommer att förbättra denna model på nästa föreläsning

Stacken

Funktioners minne hanteras av en **LIFO-stack** av **frames** — analogt med stapeln av tallrikar

Heapen

Dynamiskt data i programmet hanteras av en **heap** där data föds och dör — analogt med det rutade pappersarket

I vissa programspråk (C) måste programmeraren själv sudda på arket så att det inte blir fullt — i andra (Java) håller systemet reda på vad som kan suddas ut och lämna plats för nytt datas

Pekare

En pekare är en adress i koordinatsystemet till en plats där det ligger ett värde

En pekarvariabel i en funktion **ligger alltså på stacken** men **pekar till en plats på heapen**

Två pekarvariabler a **och** b **som innehåller samma adress är alias**

Båda tillåter att man läser eller skriver till samma plats i minnet `M`

Tilldelning till `a` eller `b` påverkar inte `M` — och bryter aliaseringen

Pekare låter oss dela värden mellan funktioner på ett disciplinerat sätt

Globala variabler är farliga eftersom de är synliga överallt — oerhört svårt att resonera om deras värden

Pekarsemantik

Tilldelning och överföring av pekarvariabler kopierar endast adressen till ett värde (billigt), inte hela värdet (dyrt)

Demo att dela värden med hjälp av pekare

Värdesemantik kontra pekarsemantik

Värdesemantik: parameteröverföring och tilldelning sker genom kopiering av värden

```
a = b; // kopiera datat i b till a
```

Pekarsemantik: parameteröverföring och tilldelning sker genom kopiering av adresser till värden

```
a = b; // låt a och b vara alias för samma data
```

För att veta huruvida en tilldelning har värdesemantik eller pekarsemantik ovan måste man veta typen på b

Strängar i C

I C är en sträng en array av heltal

Varje heltal motsvarar en bokstav enligt någon tabell för teckenkodning, t.ex. UTF-8 (ex: 97 = 'a', 65 = 'A', etc.)

För enkelhets skull begränsar vi oss till Engelska alfabetet

https://en.wikipedia.org/wiki/ASCII

En array i C är ett konsekutivt (sammanhängande) minnesutrymme av värden

En array vet inte sin längd

Två tekniker: håll reda på den i separat variabel (ex: `argc`) eller använd ett överenskommet stoppvärde (i strängar: tecknet '\0')

Strängliteraler stoppar automatiskt in ett "null-tecken" ('\0')

`char *str = "four";` har fem tecken:

'f'	'o'	'u'	'r'	'\0'

Ekvivalenta deklarationer:

```
char s[] = "Hello"
char s[] = {'H', 'e', 'l', 'l', 'o', '\0'}
char s[] = {72, 101, 108, 108, 111, 0}
```

Arrayer i C

```
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

Avsaknad av 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?

Arrayer har pekarsemantik i C

Operationen `arr1 == arr2` **betyder: ”är pekarvariablerna** `arr1` **och** `arr2` **alias till samma array?”**

För att jämföra t.ex. två strängar måste vi jämföra innehållet i dem:

```
int arr1_len = strlen(arr1);
int arr2_len = strlen(arr2);

if (arr1_len != arr2_len) return false;

for (int i = 0; i < arr1_len; i += 1) {
    if (arr1[i] != arr2[i]) { return false; }
}

return true;
```

(Se funktionen `strcmp` för hur man kan jämföra strängar)

Kopiera en sträng `a` **till en sträng** `b`**:** `strcpy(b, a)`

Duplicera en sträng `a`**:** `strdup(a)`

Lägg till en sträng `a` **i slutet på en sträng** `b`**:** `strcat(b, a)`

Sammansatta datatyper

En sammansatt datatyp är en komplex datatyp som består av flera andra datatyper

Minns Haskell's algebraiska datatyper från PKD:

```
                    -- name, year, month, day
data Anniversary = Birthday String Int Int Int
                    -- spouse name 1, spouse name 2, year, month, day
                 | Wedding String String Int Int Int

smithWedding = Wedding "John Smith" "Jane Smith" 1987 3 4
```

Enkelt att skapa och dekonstruera med pattern matching

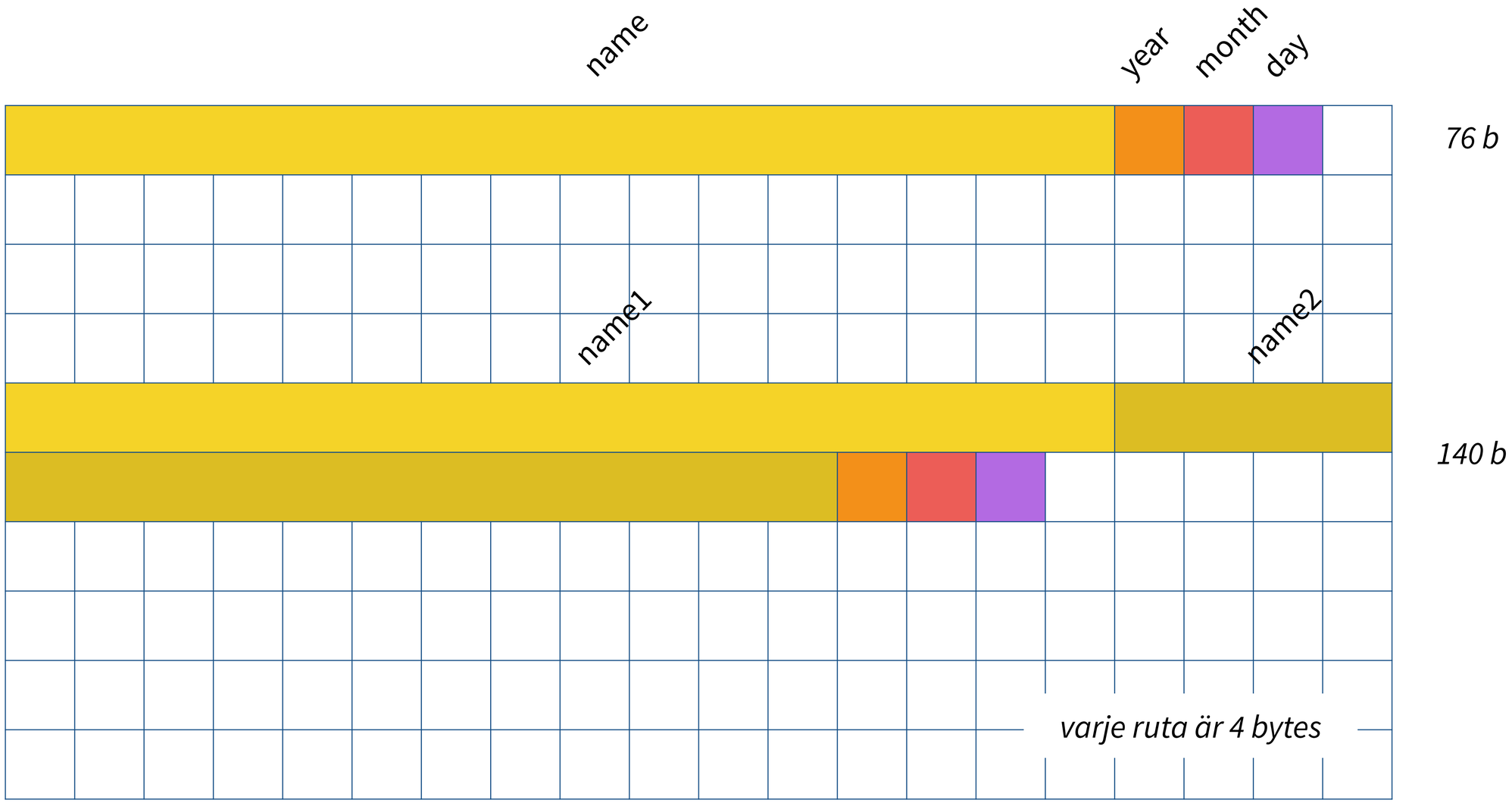
Struktar — poster i C

- Ingen pattern matching
- Inget enkelt sätt att konstruera eller dekonstruera
- Definitioner för att skapa ett sammanhängande datablock med olika *namngivna fält* som innehåller värden av *fix storlek*
- Ingen enkel motsvarighet till `Anniversary`

```
struct birthday
{
  char name[64];
  int year;
  int month;
  int day;
};
```

```
struct wedding
{
  char spouse_name1[64];
  char spouse_name2[64];
  int year;
  int month;
  int day;
};
```

En strukt är i stort sett endast en minneskarta



Inga algebraiska datatyper heller — men unioner

En union tillåter oss att gömma flera typer i en — som alternativ

Storleken (**`sizeof`**) på en union av T_1 och T_2 är $max($**`sizeof`**$(T_1),$ **`sizeof`**$(T_2))$

Metadata för att veta vilken typ ett värde har måste stoppas in för hand

Jmf. `kind` nedan

```
enum anniversary_kind { WEDDING, BIRTHDAY };

struct anniversary
{
  enum anniversary_kind kind;
  union
  {
    struct birthday b;
    struct wedding w;
  };
};
```

```
void use_anniversary(struct anniversary a)
{
  if (a.kind == BIRTHDAY)
    {
      a.b.name = ...
    }
  else
    {
      a.w.spouse_name1 = ...
    }
}
```

Recap

- Recap av koncept vi mött på labb 1 och 2
- Mentala modeller
    - Substitutionsmodellen och varför den inte räcker i en imperativ setting
    - Tallriksmodellen och rutat papper-modellen
- Värdesemantik och pekarsemantik
- Jämförelse med avseende på identitet och ekvivalens
- Strängar och arrayer
- Sammansatta datatyper
    - Poster
    - Unioner

Nästa föreläsning blir det pekare!

Väl mött på labben!