

Screencast

Sammansatta datatyper

Motivation och några exempel

- Enkla datatyper som **int**, **char*** etc. duger för att beskriva enkla värden, men data är i regel komplext och har flera beståndsdelar
- Vi kan använda flera variabler (t.ex. `size`, `price`) för att beskriva komplexa värden, men
 - Om vi har flera värden är det inte självklart t.ex. vilken storlek som hör samman med vilket pris
 - Blandar vi ihop variablerna får vi inga fel eftersom typerna är rätt
- Vi kan använda arrayer för att beskriva komplexa värden med flera beståndsdelar, men
 - Endast om de har samma typ
 - Eftersom värdena måste accessas med index måste vi trixa för att behålla läsbarhet etc.
- C tillhandahåller en mekanism för att skapa "sammansatta typer" som löser dessa problem
 - Syntaktiskt tydligt vilka värden som hör samman
 - En enda deklaraionsplats för en sammansatt typ (enkelt att modifiera)
 - Medger på ett enkelt sätt tydlig namngivning

- Här följer några kodexempel på sammansatta datatyper definierade med C:s **struct**-konstruktion:

En punkt i planet:

```
1      struct point {
2          int x;
3          int y;
4      };
```

En person i ett hypotetiskt lönesystem:

```
1      struct person {
2          char *name;
3          unsigned int salary;
4          struct person *manager;
5      };
```

Ett bankkonto med insättnings/uttagshistorik på 128 poster:

```
1      struct action {
2          int deposit; // withdrawals using negative numbers
3      };
4      struct account {
5          long balance;
6          struct action history[128];
7      };
```

- Punkten i planet har typen **struct** point och har två *fält* x och y båda av typen **int**:

```
1    struct point {  
2        int x;  
3        int y;  
4    };
```

Observera att det *inte* går att initiera fält i en ”strukt”, t.ex. **int** x = 0; går ej.

Man kan deklarera en variabel att vara av typen **struct** point så här:

```
struct point center = { 0, 0 };
```

Här initierar vi x och y till 0, men det är frivilligt. (Se även { .x = 0, .y = 0 }.)

Vill man ha åtkomst till en punkts x eller y-koordinat använder man punktnotationen:

```
center.x = 27;  
int foo = center.y;
```

Vid tilldelning mellan variabler av struct-typ kopieras samtliga värden:

```
struct point p1 = { 5, 9 };  
struct point p2 = { 1, 3 };  
p1 = p2;  
// p1.x = p2.x; // Dessa två rader är ekvivalenta med p1 = p2  
// p1.y = p2.y;
```

- Funktioner kan ha parametrar av struct-typ:

```
int sameManager(struct person p1, struct person p2) {  
    return p1.manager == p2.manager;  
}
```

- En pekare till ett värde av struct-typ är möjlig (se screencast om pekare). Detta exemplifierades med manager-fältet i typen person.

Notera en viktig skillnad vid åtkomst till fält via en variabel som är en pekare till en struct: *punkt-notationen ersätts med en pil-notation!*

```
struct person p1 = ... ; // initiering utelämnad  
... = p1.manager; // notera punkt-notation  
... = p1.manager->name; // notera pil-notationen
```

En variabel **struct** person p håller en egen kopia av en person. p.manager = ... förändrar bara personen p.

Så fort pekare kommer in i bilden blir det svårare att resonera kring vem som kan se effekterna av en tilldelning, p:s pekare till sin manager är bara p:s, men själva managern är förmodligen delad mellan flera anställda, och en förändring av dennes namn kan därför få *icke-lokala* konsekvenser.

Pil-notationen tyddliggör just detta: varenda gång en pil kommer in i programmet läses eller skrivs data som är potentiellt delat med andra!