Screencast

Typer, typomvandling, typedef

C är statiskt, manifest och svagt typat

– En variabels typ bestämmer de möjliga värden en variabel kan anta:

```
unsigned char x; // x är alltid mellan 0 och 255
struct person *p; // p pekar alltid på en sammansatt datatyp ''person''
...
```

Att C är *manifest typat* betyder att variablers (etc.) typer måste deklareras explicit av programmeraren; de räknas inte ut av kompilatorn.

– C är *statiskt typat* vilket betyder att typen för en variabel (likaså signaturen hos en funktion) *binds* vid kompilering.

Detta betyder att C har en bestämd uppfattning om vilka värden som är tillåtna att sparas i en variabel (skickas som argument till en funktion, etc.). Denna information används för att hitta felaktiga program och ger upphov till kompileringsfel.

Det finns många valida program som inte går att statiskt typa – priset för felsökningen vid kompileringen.

– C är *svagt typat*, vilket enkelt kan beskrivas som möjligheten att använda ett värde som ett annat, eller låta värden uppstå ur ingening. Följande program utnyttjar detta:

```
char s[] = "Hello, world!";
unsigned int i = *((int *)s);
```

Vad som händer i detta program är att vi tvingar C att behandla `s` som en pekare till ett heltal (normalt 4 eller 8 bytes[1]), istället för en pekare till enskilda tecken (normalt 1 byte).

Detta medför att den yttersta avrefereringen (den vänstraste `*`) läser 4 bytes, d.v.s. de 32 bitar som tillsammans bildas av `"Hell"` och tolkar dem som ett heltal. På motsvarande sätt skulle vi kunna göra `*((int *)s) = 4711` och få... Pröva!

Ett annat sätt att skapa sig ett godtyckligt heltal `j` kan vara detta:

```
int *i = malloc(sizeof(int));
int j = *i;
```

På den första raden returnerar `malloc` en pekare till 4 bytes i minnet som utgör ett heltal. De 32 bitar som råkar ligga där (vilket beror på hur just det minnet använts tidigare sedan datorn slogs på) utgör värdet på `*i`. Följaktligen har vi på rad 2 initierat `j` från ett godtyckligt heltal.

[1] För enkelhets skulle antar vi 4 här.

– Den svaga typningens problem blir extra tydligt vid användning av sammansatta typer. Ponera följande kod:

```
struct action {
  int deposit; // withdrawals using negative numbers
};
struct account {
  long balance;
  struct action history[128];
};

struct account a;
```

Deklarationen av `a` skapar ett nytt konto med okänt saldo, samt en historik med 128 in- och uttag helt baserat på vad som råkar finnas på de minnesplatser som `a` använder!

På samma sätt, pröva att ändra `balance` i `account` till en pekare till en `long` och gör `*(a.balance) = 0;`. Vi har inga garantier på vad som finns på minnesplatsen `*(a.balance)` och följaktligen kan det vara ett värde utanför det minne vårt program får läsa, varvid det skjuts ned av operativsystemet vid försök till åtkomst.

Typomvandlingar

– Föregående diskussion exemplifierade explicit typomvandling (eng. type cast). Man kan tvinga C att se ett värde som en viss typ med hjälp av en typomvandling som har syntax:

$$(T)\ exp$$

där *exp* är ett uttryck och T är en typ. Exemplet ovan:

```
(int *)s
```

talar om för C att `s` är att betrakta som en variabel som pekar på en eller flera **`int`**:ar. (Fast det i själva verket är en sträng.)

Notera att typomvandlingen enbart talar om för C hur man skall se typen på uttrycket. *Själva värdet förändras inte!* Alltså: typen på (variabeln och uttrycket) `s` är fortfarande **`char`**`*`, värdet som `s` pekar på är fortfarande strängen `"Hello, world!"` men typen på `(int *)s` är **`int`** `*`.

Typomvandlingar är ofta nödvändiga i C, speciellt när man hanterar godtyckliga data med **`void`**`*`-pekare. (Som ännu inte har introducerats här!)

Automatiska typomvandlingar

– C bjuder på automatiska typomvandlingar (eng. coercions) mellan vissa typer. Notera i koden nedan hur heltalet 42 omvandlas till 42,0.

```
double x;
x = 42; // automatisk typomvandling från 42 => 42,0
```

– Reglerna för typomvanlingar är för omfattande för att sammanfattas här, men som tumregel kan man anta att

  – I uttryck där heltal kan användas ”befordras” varje mindre heltal (t.ex. **short**, **enum**) till ett heltal
  – Alla heltal kan förvandlas till flyttal
  – När ett flyttal förvandlas till ett heltal trunkeras det, alltså `(int) 2.99` = 2.
  – Vid applikation av en binär operator till operander av typerna T_1 och T_2 där T_2 är den ”största typen”[2] konverteras T_1 till T_2.
  – En pekare kan förvandlas till en heltalstyp stor nog att rymma densamma.

[2] Bl.a. **long double** > **double** > **float** > **long long int**.

Typedef

- En typs namn kan bidra till läsbarhet och ökad abstraktion
- C tillhandahåller konstruktionen **`typedef`** för att introducera nya typnamn
- Typnamn delar samma globala namnrymd så man bör använda denna funktion med eftertanke
- Två exempel på användande av **`typedef`**:

```
typedef int kilos;
typedef struct account *Account;
```

  Det första exemplet höjer abstraktionsnivån genom att introducera `kilo` som ett alias till `int`. Om vi t.ex. håller på med en databas på ett gym kan vi anta att $0 \geq$ `kilos` ≥ 200 och omedelbart bli förvånade när vi upptäcker en variabel av typen **`kilos`** som är negativ.

  Det andra exemplet illustrerar ett mönster som vi kommer att använda löpande på kursen[3]. Vi introducerar `Account` som en typ som avser en pekare till en **`struct`** `account`.

[3]Stor bokstav på typen används för att avse en ”pekartyp”.

– Med hjälp av typedef kan vi nu skriva om definitionen

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

så här:

```
1     typedef struct person *Person;
2     struct person {
3       char *name;
4       unsigned int salary;
5       Person manager;
6     };
```

– Det går även att kombinera **typedef** direkt med en **struct**-deklaration:

```
typedef struct action {
  int deposity; // withdrawals using negative numbers
} action, *Action;
```

I detta fall deklarerar vi till och med *två* typalias: `action` istället för **struct** `action` och `Action` istället för **struct** `action *`. (Det första aliaset är dock relativt onödigt!)