# F07 - Konstanter, typer och slumptal

## Programmeringsteknik med C och Matlab, 7,5 hp

Niclas Börlin
niclas.borlin@cs.umu.se

Datavetenskap, Umeå universitet

2023-10-09 Mån



## Påminnelse

- Fönstret för att anmäla sig till tentamen stänger i morgon
- Är öppet till och med 10 okt
- Instruktioner finns på https://www.umu.se/student/mina-studier/tentamen/digital-salstentamen/



## Konstanter



## Konstanter — #define (1)

- Vissa värden används på flera ställen i programmet och kan ses som konstanter, ex:
  - `PI = 3.141592654`
  - `VAT = 0.25` (moms)
- Ett sätt att definiera konstanter är med hjälp av direktivet *#define*

```
1 #define VAT 0.25
2 #define PI 3.141592654
```

- Läggs lämpligtvis direkt efter *#include*-raderna i början på filen

## Konstanter — #define (2)

- Observera att *#define* inte deklarerar en variabel!
- Preprocessorn byter ut alla förekomster av texten `VAT` i koden mot texten `0.25`
  - Därför ska *#define*-satser inte avslutas med semikolon
- För att använda konstanterna använder vi sedan deras namn

```
area = PI * r * r;
```

vilket pre-processorn byter ut till

```
area = 3.141592654 * r * r;
```

innan koden kompileras

- *#define*-deklarerade konstanter skrivs normalt med versaler för att det skall vara tydligt att deras värden inte kan ändras



## Konstanter — #define (4)

- Tips: Använd alltid parenteser runt *#define*-uttryck!
- Koden

```
1  #include <stdio.h>
2
3  #define C 1+2
4
5  int main(void)
6  {
7        int c = C;
8        int d = C * 2;
9        return 0;
10 }
```

kommer att översättas till

```
1  #include <stdio.h>
2
3  // #define C 1+2
4
5  int main(void)
6  {
7        int c = 1+2;
8        int d = 1+2 * 2;
9        return 0;
10 }
```



## Konstanter — #define (4)

- Med parenteser runt uttrycket blir det rätt:

```
1  #include <stdio.h>
2
3  #define C (1+2)
4
5  int main(void)
6  {
7        int c = C;
8        int d = C * 2;
9        return 0;
10 }
```

```
1  #include <stdio.h>
2
3  // #define C (1+2)
4
5  int main(void)
6  {
7        int c = (1+2);
8        int d = (1+2) * 2;
9        return 0;
10 }
```



## Konstanter - const

- Ett annat sätt att deklarera konstanter är med typmodifieraren `const`

```
const float pi = 3.141592654;
```

- En `const`-deklarerad variabel kan användas precis som en vanlig variabel men den kan inte ändras
- Till skillnad från *#define*-deklarerade konstanter har en `const`-deklarerad variabel en väldefinierad räckvidd, precis som vanliga variabler
- Alla ”tal” som används på mer än ett ställe bör göras om till konstanter

## När deklarerar man konstanter?

- Matematiska konstanter
- "Globala konstanter"
  - Max storlekar på fält
- Typfall

```c
#include <stdio.h>

#define ROWS 20
#define COLS 40

#define BUFSIZE 80

int main(void)
{
    int world[ROWS][COLS];
    char buffer[BUFSIZE];
    ...
}
```



## Typer



## Varför typer?

- Om vi tittar på 64 bitar i minnet:

  `00101010 01000100 01000001 01010100`
  `01001111 01010010 00101010 00101010`
- Vad är det?
  - 2 st 32 bitars `int`?
  - En binär (tex svartvit) 8x8 bild?
  - Ett 64 bitars `double` flyttal?
  - Tecknen: `*DATOR**`?
- Typer behövs bland annat för att veta hur data i minnet ska tolkas



## Heltal

- Datatypen `int` används tillsammans med `char`, `short int`, `long int` och `long long int` för att lagra *heltal* i C
- Varje heltalsdatatyp kan vara `signed` (kan lagra både positiva och negativa värden) eller `unsigned` (bara ickenegativa)
- Matematisk sett finns det oändligt många heltal
- Med ett ändligt antal bitar får vi en gräns för hur stora heltal som kan lagras

# Heltal

- Lagras i basen 2 (*binärt*) i minnet
- Ett heltal som lagras i $n$ bitar kan anta $2^n$ olika värden
  - Om vi bara behöver positiva heltal så blir intervallet $[0, 2^n - 1]$
- Om vi också behöver negativa heltal så används en bit som teckenbit
  - Intervallet blir då $[-2^{n-1}, 2^{n-1} - 1]$
  - Hälften av intervallet är negativt, hälften är icke-negativt



# Storlek på vanliga heltal

- C-standarden beskriver bara det minsta antalet bitar som en heltalsvariabel får vara

| Typ | Bitar | Min | Max | Precision |
|---|---|---|---|---|
| `char` [1] | 8 | | | |
| `signed char` | 8 | -128 | 127 | |
| `unsigned char` | 8 | 0 | 255 | |
| `[signed] int` | 16 | -32768 | 32767 | |
| `unsigned int` | 16 | 0 | 65535 | |
| `[signed] short int` | 16 | -32768 | 32767 | |
| `unsigned short int` | 16 | 0 | 65535 | |
| `[signed] long int` | 32 | -2147463648 | 2147463647 | |
| `unsigned long int` | 32 | 0 | 4294967295 | |
| `[signed] long long int` | 64 | $-9.2 \cdot 10^{18}$ | $-9.2 \cdot 10^{18}$ | |
| `unsigned long long int` | 64 | 0 | $1.8 \cdot 10^{19}$ | |

- En `int` är idag oftast minst 32 bitar

[1] En `char` kan vara `signed` eller `unsigned`.



# Over- och underflow (1)

- Eftersom variabler har begränsad lagringskapacitet kan man drabbas av överspill (*overflow*) eller underspill (*underflow*)
- Det kan hända om man multiplicerar eller adderar två värden vars produkt inte får plats



# Over- och underflow (2)

- Koden:

```
unsigned char c1 = 254, c2 = 2;
for (int i = 0 ; i < 5 ; i++) {
    printf("<%4d %4d>\n", c1++, c2--);
}
```

skriver ut

```
<254    2>
<255    1>
<  0    0>
<  1  255>
<  2  254>
```

- Koden:

```
signed char c3 = 126, c4 = -126;
for (int j = 0 ; j < 5 ; j++) {
    printf("<%4d %4d>\n", c3++, c4--);
}
```

skriver ut

```
< 126 -126>
< 127 -127>
<-128 -128>
<-127  127>
<-126  126>
```

## Gränser för heltal

- Det går att skriva kod som anpassar sig till hur många bitar respektive datatyp har
- I filen `limits.h` så finns definitioner på hur små/stora värden man kan lagra i en heltalsdatatyp
  - Exempel på konstanter (finns några till):
    - `INT_MAX` — max för `int`
    - `INT_MIN` — min för `int`
    - `UINT_MAX` — max för `unsigned int`
    - `SHRT_MAX` — max för `short int`
    - `SHRT_MIN` — min för `short int`



## Datatypen char (1)

- Traditionellt har ett tecken lagras i en `char` (8 bitar)
- Vi kan lagra 256 olika tecken (och styrkoder)
- Traditionellt har ASCII (*American Standard Code for Information Interchange*) definierat hur heltal ska översättas till tecken
- Bokstäverna ligger ordnade alfabetiskt, versaler i en sekvens och gemener i en annan

| Tecken | A | B | ... | Z | ... | a | b | ... | z |
|---|---|---|---|---|---|---|---|---|---|
| Värde | 65 | 66 | | 90 | | 97 | 98 | | 122 |

- Specialtecken har också sina definierade värden

| Tecken | *spc* | ! | % | ( | ) | = | @ | 0 | ... | 9 |
|---|---|---|---|---|---|---|---|---|---|---|
| Värde | 32 | 33 | 37 | 40 | 41 | 61 | 64 | 48 | | 57 |

- Värdena under 32 motsvarar osynliga styrkoder, bl.a.

| Tecken | *horizontal tab* | *linefeed* | *carriage return* |
|---|---|---|---|
| Värde | 9 | 10 | 13 |



## Datatypen char (2)

- En `char` går att använda som en liten heltalstyp:

```
char c1 = 'a', c2 = 'A', c3 = 'B';
printf("<%c>", c3 - c2 + c1);
```

ger utskriften

<b>

```
#include <stdio.h>
void main(void) {
  printf("%c %d\n", 'A', 'A');
  printf("%c %d\n",  65,  65);
}
```

ger utskriften

```
A 65
A 65
```



## Tecken bortom ASCII

- ASCII definierar bara tecken för värdena 0–127
  - Det lämnade 128 platser kvar...
  - Att representera tecken utanför det engelska alfabetet har länge varit en röra
- En del standarder har använt sig de övriga 128 platserna för att representera icke-engelska tecken
  - Den mest utbredda heter ISO-8859-1/Latin 1 (≈ Windows-1252)

| Å | Ä | Ö | å | ä | ö |
|---|---|---|---|---|---|
| 197 | 196 | 214 | 229 | 228 | 246 |

- På senare år har Unicode, speciellt UTF-8, slagit igenom allt mer eftersom det har ett bredare stöd för nationella bokstäver
  - I Unicode kan ett tecken lagras i flera bytes

| Å | Ä | Ö | å | ä | ö |
|---|---|---|---|---|---|
| 195, 133 | 195, 132 | 195, 150 | 195, 165 | 195, 164 | 195, 182 |

## Tips

- Vad ska ni tänka på?
  - Undvik åäö i filnamn
  - Undvik mellanslag i filnamn
  - ANVÄND UTF-8!



## Flyttal



## Flyttal

- Det finns tre flyttalstyper: `float`, `double` och `long double`
- Variabler av denna typ kan innehålla flyttalsvärden som `0.001`, `2.0`, `3.14159` eller `6.022E23`
- Hur stor precision en variabel av `float`, `double` eller `long double` har är systemberoende
- Oftast är precisionen för en `double` bättre än för en `float` och `long double` bättre än `double`
- Enligt den vanligaste standarden, IEEE 754 ("IEEE-standarden för flyttal"), gäller följande

| Typ | Bitar | Min | Max | Precision |
|---|---|---|---|---|
| `float` | 32 | $1.2 \cdot 10^{-38}$ | $3.4 \cdot 10^{38}$ | 7 sign. siffror |
| `double` | 64 | $2.2 \cdot 10^{-308}$ | $1.8 \cdot 10^{308}$ | 15 sign. siffror |
| `long double` | 80 | $3.4 \cdot 10^{-4932}$ | $1.2 \cdot 10^{4932}$ | 18 sign. siffror |



## Flyttalskonstanter

- Vi kan skriva heltal som flyttalskonstanter i vår källkod om vi skriver på formen `1.0` och `2.0`
- Texten `3` definerar däremot en heltalskonstant
- Flyttal kan också anges som `1.234567e5` vilket motsvarar $1.234567 \cdot 10^5$
- Ett suffix kan läggas till som anger vilken typ konstanten är
- Suffixet `f` eller `F` anger att konstanten är `float`
- Suffixet `l` eller `L` anger att konstanten är `long double`
- Ett flyttal utan suffix är av typen `double`
- Konstanten `3.7F` är en flyttalskonstant av typen `float`

## Ta reda på storlekar i kod

- Koden

```c
#include <stdio.h>
#include <float.h>

int main(void)
{
  printf("DBL_MAX = %e\n", DBL_MAX);
  printf("DBL_MIN = %e\n", DBL_MIN);
  printf("FLT_MAX = %e\n", FLT_MAX);
  printf("FLT_MIN = %e\n", FLT_MIN);
  return 0;
}
```

ger utskriften

```
DBL_MAX = 1.797693e+308
DBL_MIN = 2.225074e-308
FLT_MAX = 3.402823e+38
FLT_MIN = 1.175494e-38
```



## Problem

- Cancellation error
  - Ett stort tal adderas till ett mycket litet
- Aritmetisk underflow
  - Operationer som involverar små tal eller små skillnader
  - Två små flyttal multipliceras
- Aritmetisk overflow
  - Operationer som involverar stora tal
  - Ex: Två stora flyttal multipliceras



## Exempel

- Att reella tal inte lagras exakt kan ge upphov till problem
- Om vi antar att vår dator kan lagra en `float` med 5 signifikanta siffror och att datorn använder basen 10 (för att det inte ska bli allt för många nollor i exemplet)

```c
float x = 12345.0F;
float y = 1e-1F;
float z = 1.0 / (x + y - 0.12345e5F); /*problem*/
```

- När uttrycket i nämnaren skall evalueras måste exponenten av de tre talen vara av samma storlek

```
x = 0.12345e5
y = 0.00000e5
x + y - 0.12345e5 == 0
```

- Hur hade vi kunnat lösa problemet?



## Ändlig precision

- Koden

```c
#include <stdio.h>

int main(void)
{
    float f = 2;
    double d = 2;
    for (int j = 0 ; j < 50 ; j++) {
        f = f / 7;
        d = d / 7;
    }
    for (int j = 0 ; j < 50 ; j++) {
        f = f * 7;
        d = d * 7;
    }
    printf("\nfloat: %.18f\ndouble: %.18f\n", f, d);
    return 0;
}
```

ger utskrift

```
float: 2.001028299331665039
double: 1.999999999999999556
```

# Slumptal



# (Pseudo)slumptal (1)

- Funktionen `rand()` ger ett "slumptal" mellan 0–`RAND_MAX`
  - `RAND_MAX` kan vara 32767 eller 2147483647 beroende på plattform (dator, operativsystem)
- `rand()` returnerar ett nytt värde för varje anrop
  - Kör vi koden

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

int main(void)
{
  int i = rand();
  int j = rand();
  return 0;
}
```

    kommer `i` och `j` troligen inte att vara samma tal
- Vill vi ha ett värde mellan `1` och `n` kan vi göra så här:

```c
int rnd = rand() % n + 1;
```



# (Pseudo)slumptal (2)

- Funktionen `rand()` är en s.k. pseudoslumptalsgenerator (*pseudo random number generator*)
  - Den ger en talsekvens som verkar slumpmässig men inte är det
- Funktionen `rand()` arbetar utifrån ett så kallat *frö* (*seed*)
  - Värdet på fröet sätts med funktionen `srand()`
  - Varje värde på fröet ger en egen sekvens av "slumptal"
- Koden

```c
#include <stdio.h>
#include <stdlib.h>
int main(void)
{
    srand(28);
    int i = rand();
    int j = rand();
    printf("i=%d, j=%d\n", i, j);
    return 0;
}
```

    kommer att ge samma värden på `i` och `j` vid varje körning (`i=1110582131, j=96515849` på min dator)



# (Pseudo)slumptal (3)

- Vill vi få olika "slump"-sekvenser olika gånger vi kör programmet kan vi t.ex. göra så här:

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

int main(void)
{
  // Use number of seconds since 1970-01-01 00:00:00 as seed
  srand(time(NULL));
  int i = rand();
  ...
  return 0;
}
```

# Exempel

- Simulera 10000 kast med två tärningar
  - Räkna hur ofta vi får respektive kombination av tärningar
  - Spara resultatet i en tvådimensionell array
  - Skriv ut resultatet när det är klart
- Algoritm:
  - Initiera arrayens element till 0
  - Upprepa 10000 gånger
    - Slå två tärningar
    - Öka värdet på arrayelementet som ges av de två tärningarnas värden med 1
  - Skriv ut resultatet