

F05 - Funktioner

Programmeringsteknik med C och Matlab, 7,5 hp

Niclas Börlin

niclas.borlin@cs.umu.se

Datavetenskap, Umeå universitet

2023-10-05 Tor

- ▶ Ett mål för mjukvaruutvecklare är förstås **felfri kod**
- ▶ En bra strategi är att **återanvända** kod som redan utvecklats och testats utförligt
- ▶ I språket C ingår många färdiga s.k. **biblioteksrutiner**, t.ex. de matematiska funktionerna `sin`, `cos`, `exp` (e^x), `sqrt` (\sqrt{x}), etc.

Exempel

```
1  #include <stdio.h>
2  #include <math.h>
3
4  int main(void)
5  {
6      double x, result;
7
8      // Read a number from the user
9      printf("Enter a number: ");
10     scanf("%lf",&x);
11
12     // Compute the sine of x
13     result = sin(x);
14
15     // Output the result
16     printf("sin(%f) = %f\n", x, result);
17
18     printf("\nNormal exit.\n");
19     return 0;
20 }
```

Header-filer

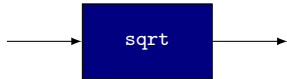
- ▶ Filen `math.h` är en s.k. **header-fil** och innehåller **deklarationen** av funktionen `sin()`, m.fl.
- ▶ **Definitionerna** av funktionerna finns i färdigkompilerad form på **annan plats** i systemet
- ▶ Mer om biblioteksrutiner (standardfunktioner) finns i kapitel 10 i kursboken

Top-down design

- ▶ En av de grundläggande designmetoderna kallas *Top-down*
 - ▶ Den går ut på att man bryter ner ett problem i flera **delproblem**
 - ▶ Sen **löser** man delproblemen, ett och ett
 - ▶ Eventuellt genom att dela upp dem i **ännu mindre** delar, etc.
- ▶ Ett viktigt verktyg för *top-down design* i C är **funktioner**

Funktioner

Funktioner (1)

- ▶ En funktion går att se som en svart "**låda**":
 - ▶ `y = sqrt(x);`

- ▶ Vi behöver inte veta hur `sqrt`-lådan **fungerar**
- ▶ Vi måste däremot veta
 - ▶ vad den **heter**,
 - ▶ vad den **gör**,
 - ▶ vilket **indata** den vill ha och
 - ▶ vad den returnerar för **utdata**

Funktioner (2)

- ▶ Funktioner gör det möjligt att
 - ▶ lösa komplexa problem genom att **dela upp** koden i logiska delar
 - ▶ skriva program som är lättare att **förstå**
 - ▶ skriva program som har delar som är lätta att **byta ut**, tex vid framtida uppgraderingar
 - ▶ **samarbeta** med andra i utvecklingsarbete
- ▶ **Upprepad kod** ("klipp-och-klistra") hör antagligen hemma i en funktion

Funktioner i C

- ▶ En funktion i C är kod som
 - ▶ har ett **namn**,
 - ▶ tar ett specifikt antal **parametrar** (≥ 0), var och en av en specificerad **typ**,
 - ▶ "gör något" och
 - ▶ eventuellt **returnerar** ett **värde** av en specifik **typ**
- ▶ Exempel:

```
1 int add(int n, int m)
2 {
3     int sum;
4     sum = n + m;
5     return sum;
6 }
```

Ytterligare exempel

```
1 double sphere_volume(double radius)
2 {
3     double volume;
4     volume = 4 * M_PI; // from math.h
5     volume = volume * radius * radius * radius;
6     volume = volume / 3;
7     return volume;
8 }
```

Funktionsdeklarationer

- ▶ Det vi nu sett är två exempel på funktions**definitioner**
 - ▶ Vi kan också **deklarera** funktioner utan att **definiera** dem
- ```
int add(int n, int m);
double sphere_volume(double radius);
```
- ▶ En funktionsdeklaration, eller **funktionsprototyp**, kan ses som ett **löfte**:
    - ▶ definitionen kommer att finnas någon annanstans
  - ▶ Notera att funktionsdeklarationen avslutas med ett **semikolon** där annars definitionen skulle vara

## Funktionsanrop

- ▶ När vi deklarerat en funktion kan vi **använda** den
- ▶ Det kallas att **anropa** funktionen
- ▶ Exempel:
  - ▶ `x = add(5, 3);`
  - ▶ Funktionen add anropas med **parametervärdena** 5 och 3
  - ▶ Anropet kommer att **returnera** ett värde (förhoppningsvis 8) som **lagras** i variabeln x
- ▶ Parametrarna kan vara vilka **uttryck** som helst, t.ex. värdet hos en variabel eller ett aritmetiskt uttryck

```
x = add(n, 3);
```

- ▶ Det värde som skickas till funktionen är **värdet** som är lagrat i variabeln **n**, inte variabeln själv

## Returvärden

- ▶ Returvärdet från en funktion kan användas i ett **uttryck**
  - ▶ Returvärdet kan också **ignoreras**
- ▶ Det är vanligast att tilldela en **variabel** returvärdet
- ▶ För att tilldelningen ska vara giltig måste **funktionens** returtyp vara **kompatibel** med **variabelns** typ

```
1 double sphere_volume(double radius);
2
3 int main(void)
4 {
5 double b;
6 int i;
7
8 b = sphere_volume(2.0); // Will be stored "exactly" as 2.094395...
9
10 i = sphere_volume(2.0); // Will be truncated to 2
11
12 sphere_volume(2.0); // The return value will be ignored
13
14 return 0;
15 }
```

- ▶ Att två typer är **kompatibla** betyder att typerna är **samma** eller att det går att **konvertera** (översätta) från ena typen till den andra

## Typen void

- ▶ Det går att skriva funktioner som **inte** tar någon **indata** och/eller **inte returnerar någonting**
- ▶ I C löser man det genom att använda typen **void** som betyder ungefär "ingenting"

```
void hello(void)
{
 printf("Hello, World!\n");
}
```

## Generell syntax för en funktion

```
returtyp funktionsnamn(typ1 namn1, ..., typn namnn)
{
 lokala variabeldeklarationer
 beräkningar
 eventuellt returnera något
}
```

## Formella och aktuella parametrar

- ▶ I en funktionsdeklaration och -definition anger vi **formella** parametrar till en funktion
  - ▶ Det är namnen vi använder **inuti** funktionen
- ▶ De parametrar som skickas till funktionen vid anrop kallas **aktuella** parametrar eller **argument**
- ▶ De aktuella parametrarna är **uttryck**
  - ▶ Uttrycken evalueras **innan** anropet sker
- ▶ Vilken aktuell parameter som knyts till vilken formell parameter styrs enbart av **ordningen** för parametrarna

## Lokala variabler och parameteröverföring

## Kodexempel med funktionsanrop

- Här är ett program med två funktioner, `print()` och `add()`

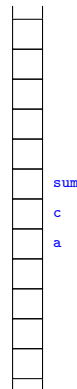
```
code/vars-on-stack.c
1 #include <stdio.h>
2 void print(int v)
3 {
4 printf("v = %d\n", v);
5 }
6 int add(int c, int d)
7 {
8 int a;
9 a = c + d;
10 return a;
11 }
12 int main(void)
13 {
14 int a = 2;
15 int c = 3;
16 int sum;
17 sum = add(a, c);
18 print(sum);
19 sum = add(sum, c + 4);
20 print(sum);
21 return 0;
22 }
```

## Lokala variabler och stacken (1)

- Lokala variabler lagras i en minnesarea som kallas **stacken**

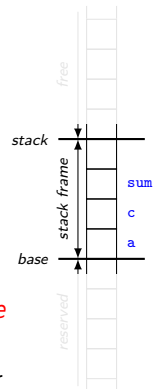
```
code/vars-on-stack.c
12 int main(void)
13 {
14 int a = 2;
15 int c = 3;
16 int sum;
```

- När en funktion anropas så **reserveras/allokeras** minne för variablerna **automatiskt**
- Vid återhopp så **frigörs/deallokeras** minnet automatiskt



## Lokala variabler och stacken (2)

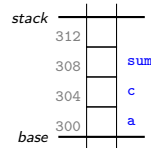
- Internt används två pekare (**base** och **stack**) för att hålla reda på funktionens del av stacken
  - Området mellan pekarna kallas för **stack frame** (aktiveringspost)
  - Området under kan betraktas som **upptaget**
  - Området ovanför kan betraktas som **ledigt**
- När funktionen körs är endast minnet **inom stack frame** åtkomligt
  - Det är en av vinsterna med funktioner, att förändringar kan endast göras lokalt (**lokalitet**)



## Lokala variabler, minne (1)

```
code/vars-on-stack.c
12 int main(void)
13 {
14 int a = 2;
15 int c = 3;
16 int sum;
```

- ▶ Varje variabel ligger lagrad på en **adress** i minnet:
- ▶ Här ligger
  - ▶ a lagrad på adressen 300
  - ▶ c på adress 304,
  - ▶ sum på adress 308 och
  - ▶ *base*-pekaren har värdet 300
  - ▶ (adress 312 är reserverad — mer sen)
- ▶ Kompilatorn översätter **variabelreferenser** i källkoden till **minnesreferenser** i maskinkoden
  - ▶ Variabeln a är känd som (*base*+0)
  - ▶ Variabeln c är känd som (*base*+4)
  - ▶ Variabeln sum är känd som (*base*+8)



## Lokala variabler, minne (2)

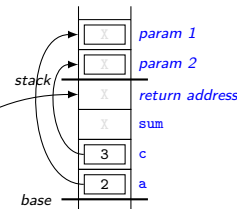
- ▶ Mängden minne som kompilatorn reserverar till en variabel bestäms av dess **typ**
  - ▶ En **int** tar vanligen 4 bytes
  - ▶ En **char** tar vanligen 1 byte
  - ▶ En **double** tar vanligen 8 bytes
- ▶ Jag kommer att ignorera det i mina skisser om det inte är viktigt
- ▶ Kom ihåg: Ni behöver hålla reda på
  - ▶ variabelns **namn**
  - ▶ variabelns **typ**men inte
  - ▶ variabelns adress
  - ▶ variabelns storlek

## Funktionsanrop (1)

- ▶ Stacken används också för **parameteröverföring** vid **funktionsanrop**

```
code/vars-on-stack.c
12 int main(void)
13 {
14 int a = 2;
15 int c = 3;
16 int sum;
17 sum = add(a, c);
```

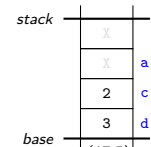
- ▶ I verkligheten lagras en **returadress** som anger var i minnet nästa kodinstruktion ligger
  - ▶ Jag kommer att skriva (*kodrad*) i stället för returadressen
- ▶ Koden med funktionsanropet på rad 17 översätts i princip till
  - 17.1 store (17.5) at (stack-4) // ret addr
  - 17.2 store (base+4) at (stack+0) // c -> p2
  - 17.3 store (base+0) at (stack+4) // a -> p1
  - 17.4 call add
  - 17.5



## Funktionsanrop (2)

- ▶ Den anropade funktionen (add()) **justerar stackpekarna...**

```
code/vars-on-stack.c
6 int add(int c, int d)
7 {
8 int a;
9 a = c + d;
10 return a;
11 }
```

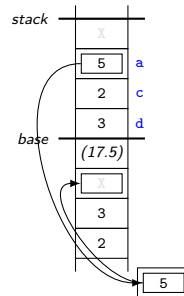


- ▶ ...och reserverar en **egen stack frame** i den fria delen av stacken
- ▶ Den anropande funktionens variabler blir **osynliga** och skyddas
- ▶ Notera att parametrarna c och d fungerar som **lokala variabler**
  - ▶ Parametrarna är **initierade** (har giltiga värden) när funktionen startar
  - ▶ Övriga lokala variabler har ett **odefinerat** värde (X i stacken)

## Återhopp

- ▶ Vid återhopp från en funktion sker följande:
  - ▶ **returvärdet** lagras i ett s.k. **register** i CPU:n,
  - ▶ **stack frame** **återställs**...

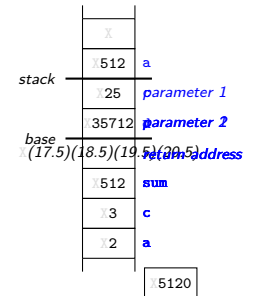
```
code/vars-on-stack.c
6 int add(int c, int d)
7 {
8 int a;
9 a = c + d;
10 return a;
11 }
12 int main(void)
13 {
14 int a = 2;
15 int c = 3;
16 int sum;
17 sum = add(a, c);
```



- ▶ ...och exekveringen fortsätter vid återhopsadressen  
17.5 store register at (base+8) // sum

## Exemplet steg för steg

```
code/vars-on-stack.c
1 #include <stdio.h>
2 void print(int v)
3 {
4 printf("v = %d\n",v);
5 }
6 int add(int c, int d)
7 {
8 int a;
9 a = c + d;
10 return a;
11 }
12 int main(void)
13 {
14 int a = 2;
15 int c = 3;
16 int sum;
17 sum = add(a, c);
18 print(sum);
19 sum = add(sum, c + 4);
20 print(sum);
21 return 0;
22 }
```



## Variablers räckvidd (1)

- ▶ En variabels **räckvidd** (*scope*) är det **område** i källkoden där vi kan **referera** till variabeln
- ▶ En variabel som deklareras i ett block är **lokal** i det blocket
  - ▶ Block = ett antal satser inom {}
- ▶ Den deklarerade variabeln gäller från deklarationen och **nedåt** i blocket

```
1 int main(void)
2 {
3 int i = 4; // i is visible in main from here on down
4 if (i > 3) {
5 int j = 3;
6 int k = i + j; // We can access j here...
7 }
8 int m = i + j; // ...but not here
9 return 0;
10 }
```

- ▶ Om vi refererar till en variabel **utanför** dess räckvidd kommer kompilatorn att ge ett felmeddelande
  - ▶ Ovanstående kod ger kompileringsfel på rad 8

## Variablers räckvidd (2)

- ▶ Om samma variabelnamn använts på flera nivåer gäller den deklaration som ligger **närmast** det block där variabeln används
  - ▶ Den mest **lokala** deklarationen döljer alltså ut den mer globala

```
1 #include <stdio.h>
2 int main(void)
3 {
4 int i = 4;
5 if (i > 3) {
6 int i = 3; // this will hide the "outer" i
7 int k = i;
8 }
9 return 0;
10 }
```

- ▶ Det är **dålig programmeringsstil** att återanvända variabelnamn som döljer andra

## Variablers räckvidd (3)

- ▶ Om en variabel deklarerar **utanför** alla block i början av filen blir den **global**
- ▶ Globala variabler ska **undvikas**
  - ▶ Ger kod som är svår att **underhålla**
  - ▶ Gör det svårare att skriva **modulär** kod
  - ▶ Gör det svårt att skriva **återanvändbar** kod
- ▶ Det är oftast **dålig programmeringsstil** att använda globala variabler
  - ▶ Ni ska bara använda globala variabler om det finns **väldigt bra skäl** för det
  - ▶ Händer inte på denna kurs

## Placering av main()

- ▶ Om en källkodsfil innehåller **flera funktioner** förutom main() så finns det flera varianter på i vilken ordning man kan placera funktionerna
- ▶ Vi kommer att begära att ni använder denna variant, där funktionerna är definierade **innan** dom används

```
1 #include <stdio.h>
2 int sum(int c, int d)
3 {
4 return c + d;
5 }
6 int diff(int a, int b)
7 {
8 return a - b;
9 }
10 int main(void)
11 {
12 int a = 2, b = 1;
13 printf("%d + %d = %d\n", a, b, sum(a, b));
14 printf("%d - %d = %d\n", a, b, diff(a, b));
15 return 0;
16 }
```

## Funktioner med variabelt antal parametrar

- ▶ Det är möjligt att skriva funktioner som tar ett **variabelt** antal parametrar
- ▶ Vi kommer att begränsa oss till att skriva funktioner som tar ett **bestämt** antal parametrar
- ▶ Vi kommer dock att **använda** några vanliga funktioner som tar ett variabelt antal, t.ex. `printf()`, `scanf()`, etc.