Kapitel 7

Funktioner

7.1 Exempel på en funktion

Det finns två typer av funktioner inom programmering:

- **Funktioner med returvärde**
- **Funktioner utan returvärde, s.k. void-funktioner**

I detta avsnitt behandlar vi den första typen. Funktioner utan returvärde, s.k. **void**-funktioner kommer att tas upp senare (sid 144). Låt oss gå tillbaka till sid 64:

```
// Hour2Sec.cpp
// Läser in tiden i timmar, minuter och sekunder, omvandlar
// allt till sekunder och skriver ut resultatet
#include <iostream>
using namespace std;

int main()
{
   int  tim, min, sek, totalsek;

   /* I n m a t n i n g */
   cout << "\nGe timmar, minuter, sekunder "
        << "skilda med mellanslag:  ";
   cin  >> tim >> min >> sek;

   /* B e a r b e t n i n g */
   totalsek = 3600*tim + 60*min + sek;

   /* U t m a t n i n g */
   cout << '\n' << tim << " timmar, "<< min<< " minuter och "
        << sek << " sekunder är " << totalsek
        << " sekunder totalt.\n\n";
}
```

Vi hade då av en viss anledning delat in koden ovan i strukturen *inmatning – bearbetning – utmatning*, vilket vi nu ska återkomma till. Vi vill nämligen separera *bearbetningen* från **main()** och skriva den som en funktion. Så här skulle en sådan funktion se ut:

```
int totalsek(int t, int m, int s)
{
   /* B e a r b e t n i n g */
   return 3600*t + 60*m + s;
}
```

Första raden kallas för funktionens *huvud*, innehållande funktionens namn **totalsek()**, funktionens *parameterlista* **(int t, int m, int s)** och returvärdets datatyp **int**, kallad *returtypen*. Inom måsvingarna som avgränsar funktionen som ett block står funktionens *kropp*. Den i sin tur består av en enda sats som inleds med det reserverade ordet **return**, kallad **return**-sats, som returnerar *returvärdet*.

Uttrycket efter **return** beräknar totalsekunderna med hjälp av *parametrarna* t, m och s. Att vår funktion **totalsek()** innehåller en **return**-sats är anledningen till varför den är en *funktion med returvärde*, dvs den första av de två typer som nämndes inledningsvis. Alla dessa begrepp kommer att preciseras allt eftersom.

Funktionen **totalsek()** kan kompileras när den infogas i ett projekt i Visual Studio. Men den kan inte exekveras därför att den inte är ett *program:* **main()**, exekveringens startpunkt finns inte. Vi måste lägga in den i ett program med **main()**.

Så här gör vi för att bygga in funktionen **totalsek()** i ett program med **main()**:

```
// MyFirstFct.cpp
// Bearbetningen har flyttats till funktionen totalsek() som
// placeras före main() och anropas från main().
// In- och utmatning görs fortfarande i main().
#include <iostream>
using namespace std;
/*********************************************************/
int totalsek(int t, int m, int s) // Definition av funktion
{
   /* B e a r b e t n i n g */
   return 3600*t + 60*m + s;
}
/*********************************************************/
int main()                          // Här startar exekveringen
{
   int tim, min, sek;

   /* I n m a t n i n g */
   cout << "\n\tGe timmar, minuter, sekunder "
        << "skilda med mellanslag:  ";
   cin  >> tim >> min >> sek;

   /* U t m a t n i n g */
   cout << "\n\t" << tim << " timmar, " << min << " minuter "
        << "och " << sek << " sekunder är "
        << totalsek(tim, min, sek) << " sekunder totalt.\n";
}          // Anrop av funktion
```

Nu kan vi både kompilera och exekvera programmet ovan. Vid exekveringen anropas funktionen **totalsek()** i den sista raden av **main()**. Vad betyder det?

Vad händer när en funktion anropas?

Tre saker händer när funktionen **totalsek()** anropas från **main()**:

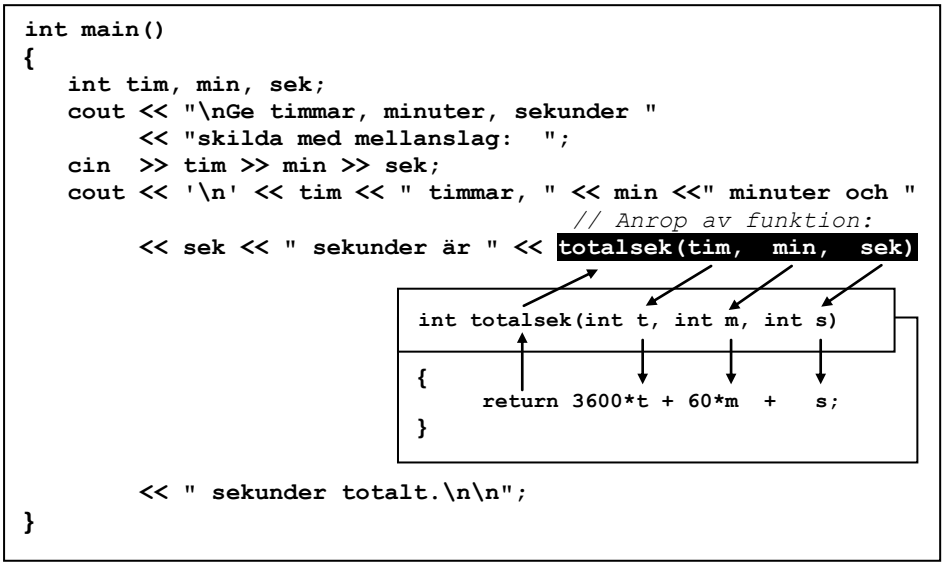
1. **Parameteröverföring** Då överförs de aktuella parametrarna **tim**, **min**, **sek**:s värden som lästs in före anropet, till de formella parametrarna **t**, **m**, **s**. Det finns olika mekanismer för parameteröverföring beroende på parameterns datatyp vilket tas upp senare. I vårt exempel kopieras värdena från **tim**, **min**, **sek** till **t**, **m**, **s**. För de aktuella parametrarna har genom definitionssatsen

`int tim, min, sek;` tre minnesceller reserverats i början av `main()`. För de formella parametrarna har tre andra minnesceller reserverats i `totalsek()`:s parameterlista via definitionerna `int t, int m, int s`. Observera att korresponderande parametrar borde vara av samma datatyp. Om vi t.ex. matar in `5 35 49` när programmet körs, tilldelas dessa värden variablerna `tim, min, sek`. Funktionsanropet vidarebefordrar värdena till funktionens formella parametrar `t, m, s`. Så hamnar de inmatade värdena i funktionskroppen till `totalsek()`.

2. **Exekvering av funktionskroppens kod** vilket i vårt fall innebär att `return`-satsen utförs dvs uttrycket `3600*t + 60*m + s` beräknas. Med inmatningen från punkt **1** blir det `3600*5 + 60*35 + 49`. Resultatet `20149` tilldelas funktionsnamnet `totalsek`. Mer finns inte just i vårt exempel.

3. **Överföring av returvärdet** sker i omvänd riktning jämfört med parameteröverföringen, nämligen från den anropade funktionen `totalsek()` till den anropande funktionen `main()`. Vi får tillbaka returvärdet från funktionen, i exemplet är det variabeln `totalsek`:s värde dvs strängen `20149` med inmatningen från punkt **1**. Att returvärdet hamnar i `main()` beror på att anropet görs med funktionsnamnet `totalsek` som är identiskt med returvärdet.

Det som händer vid anrop av en funktion är alltså att data byts ut mellan den anropande och den anropade funktionen, i vårt exempel mellan `main()` och `totalsek()`, att koden i den anropade funktionen utförs när anropet inträffar, samt att returvärdet till sist hamnar i den anropande funktionen. En översikt över dataflödet mellan dessa två funktioner (se pilarna nedan) och framför allt i *vilken ordning* deras koder utförs ges i följande bild som illustrerar punkterna **1-3** ovan:



Bilden ovan visar vad som *händer* när programmet **`MyFirstFct`** (sid 128) exekveras: Funktionen **`totalsek()`**:s kod hämtas och utförs (på bilden: klistras in) på det stället där funktionens anrop står (på bilden: framhävt med svart bakgrund). Det är skillnad mellan kod och handling. Bättre sagt: ordningen i koden – låt oss kalla det *dokumentationen* – är annorlunda än det som händer när koden körs – låt oss kalla det *aktionen*. Aktionen börjar som vanligt med exekveringen i **`main()`** och fortsätter tills den kommer till anropet av funktionen. Punkterna **1-3** ovan utförs vilket visas bl.a. med pilarna. Sedan fortsätter aktionen med den kod i **`main()`** som står efter anropet. **`main()`**:s och **`totalsek()`**:s aktioner är nästlade i varandra. Dokumentationen har en helt annan struktur. De båda funktionernas koder är inte alls nästlade i varandra, snarare tvärtom, de är isolerade från varandra. Och så måste det vara: Koden till funktionen **`totalsek()`** får inte under några omständigheter skrivas *i* **`main()`**, den måste stå *utanfö*r **`main()`**, antingen *före* eller *efter* **`main()`**, ja den kan t.o.m. ligga i en *separat* fil. Men när vi kör programmet i sin helhelt händer saker och ting i den ordning som visas på bilden.

Bilden på förra sidan visar också dataflödet som går från de aktuella parametrarna **`tim`**, **`min`**, **`sek`** till de formella parametrarna **`t`**, **`m`**, **`s`**. Dessa bearbetas i funktionen **`totalsek()`** dvs **`return`**-uttrycket beräknas. Sedan skickas resultatet av beräkningen som returvärde via funktionsnamnet till **`main()`**. Vid exekvering utförs funktionskroppens kod, precis på det stället där funktionsanropet i **`main()`** förekommer. Vid denna process spelar funktionshuvudet **`int totalsek(int t, int m, int s)`** rollen av ett *gränssnitt* mellan **`main()`** och **`totalsek()`**. Det är där kommunikationen mellan dessa separata moduler äger rum. Därför har vi satt den i en extra ruta för att understryka denna roll. Huvudet är nämligen åtkomligt både från **`main()`** och **`totalsek()`**. De skulle annars inte kunna kommunicera med varandra då de ligger i olika block. Bilden ska nämligen även visa att funktionsanropet resulterar i en *blockstruktur* som återges av ramarna i bilden. Motsvarande kod till denna blockstruktur utgörs av klamrarna **`{ }`** till både **`main()`** och **`totalsek()`**. Klamrarna bildar dessa modulers fasta gränser för kodens giltighet eller räckvidd. För att överskrida dem måste vissa regler beaktas vilket vi kommer att precisera senare. Blockstrukturen är den egentliga orsaken till att funktion **`totalsek()`** inte kan kommunicera med **`main()`** annat än via funktionshuvudet som gränssnitt. Det motiverar dessutom varför funktionerna inte får nästlas i varandra utan måste kodas separat. Mer om blockstruktur kommer att tas upp på sid 148. Inledningsvis hade vi nämnt block på sid 96.

Ett körexempel av programmet **`MyFirstFct`** kan se ut så här:

```
Ge timmar, minuter, sekunder skilda med mellanslag:  5 35 49

5 timmar, 35 minuter och 49 sekunder är 20149 sekunder totalt.
```

Placering av funktioner

När vi pratar om funktioners placering menar vi placeringen av deras *definition*. Kan man definiera funktioner var som helst i ett C++ program? I vårt fösta exempel (sid 127) placerade vi funktionen **`totalsek`** före **`main()`**. Det är också den naturliga platsen för en funktion så länge man håller på med att utveckla och testa den. Men nu, när vi är klara med utvecklingsstadiet, vill vi titta på alternativ. I större program brukar man skriva och testa sina funktioner en i taget. När man är klar med dem, vill man helst lägga dem ”åt sidan” och få upp, när man öppnar programfilen, högst på sin skärm endast den funktion som man aktuellt håller på med att utveckla och testa. ”Åt sidan” innebär antingen efter **`main()`** eller i en separat fil, vilket behandlas i de efterföljande avsnitten. Men innan vi går vidare ska vi konstatera ett absolut förbud om funktioners placering:

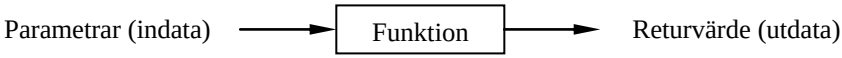
> En funktions definition får inte placeras inuti en annan funktion.

7.2 Funktionsbegreppet i programmering

Begreppet *funktion* härstammar från matematiken. Man har en formel som beräknar ett värde, säg y, utgående från ett annat, säg x, och skriver y = f(x). y är en funktion av x. Denna matematiska syn på funktion har tagits över till programmering som ett underliggande koncept och historisk utgångspunkt. Men under tiden har begreppet vidareutvecklats och fått en bredare tolkning då den tillämpats på all datoriserad problemlösning. Inom programmering inkluderar funktioner matematiska problem utan att vara begränsade till sådana.

> I ett datorprogram är en funktion ett antal satser som definieras som en separerat modul utanför **`main()`**. De utförs när funktionen anropas. Vid anropet kan funktionen ta emot indata, s.k. parametrar, bearbeta dem och returnera utdata, s.k. returvärde.

En funktion kan även jämföras med en (”svart”) låda i vilken man stoppar in indata och får ut utdata: Indata kallas även *parametrar* och utdata *returvärde*:



En funktion kan ha 0, 1 eller flera parametrar. Den kan ha 0 eller 1 returvärde. En funktion kan inte ha flera returvärden. Både parametrarna och returvärdet kan vara tal, tecken, strängar, sanningsvärden eller andra datatyper. Funktionen bearbetar de inkommande parametrarna och returnerar endast *ett* värde eller inget alls. En funktion utan returvärde kallas för **`void`**-funktion. Som separat och namngiven modul kan en funktion anropas i **`main()`**, men även även i andra program. I denna bemärkelse är en funktion ett *underprogram* (eng. *subroutine*).

”Svart” är lådan så länge vi inte själva definierat funktionen och därför inte vet hur den fungerar ”inuti”. Vi använder den endastför att lösa ett visst problem. Det gäller t.ex. för de biblioteksfunktioner som vi hittills använt i våra programexempel: **`sizeof()`**, **`_getch()`**, **`rand()`**, **`srand()`** och **`time()`**. De är förprogrammerade och lagras i biblioteksfiler. Vi behöver bara inkludera dem i våra program för att dra nytta av deras funktionalitet utan att behöva veta hur de i detalj är konstruerade. Biblioteken i alla programmeringsspråk består av sådana ”svarta” lådor.

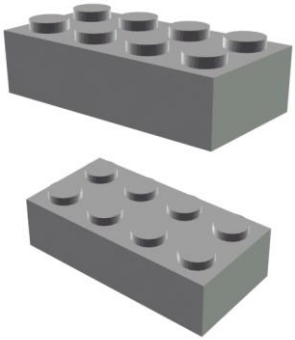
Den enda funktion som vi hittills definierat själva – och det har vi gjort i *alla* våra programexempel – är **`main()`**. Den är unik därför att utan **`main()`** som programmets exekveringspunkt händer ingenting (sid 34). I alla procedurala programspråk bildar funktioner programmets byggstenar (moduler). Att C++ som är objektorienterat även bygger på funktioner och inte bara på klasser, beror på språkets rötter i C som är ett proceduralt, men inte objektorienterat språk. C# och Java t.ex. bygger endast på klasser som inkapslar sina funktioner och döper dem, när de definieras i klasser, till *metoder*.

Varför funktioner?

Frågan är berättigad för nybörjare: Kan man inte helt enkelt skriva kod rakt ned i `main()`? Är detta med funktioner inte att krångla till det hela?

Föreställ dig en verksamhet som växer med tiden, ett expanderande företag eller en organisation med stigande antal medlemmar. Hur organiserar man jobbet? Man gör arbetsdelning. Man delegerar uppgifterna. Var och en får en väl definierad arbetsuppgift. Annars skulle man inte kunna klara av jobbet. Samma sak gör man med program vars kod växer. Det händer när man utvecklar programmet efter behov och det bara blir större och större. Man delar upp det stora programmet i mindre moduler för att kunna klara av komplexiteten. Hur det görs ska vi nu diskutera under rubrikerna: *Modularisering*, *återanvändning av kod* och *strukturering av program*.

Modularisering eller Lego-principen

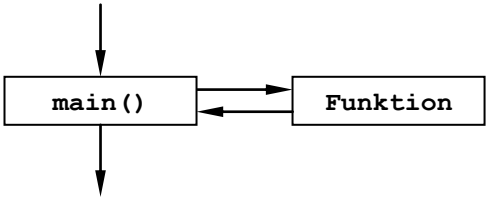


De flesta har väl någon gång som barn, eller tillsammans med sina barn, byggt ett hus, en bil eller liknande med Lego-bitar. Efter ett tag har huset kanske rasat och nya tekniska underverk har konstruerats. Men även de har någon gång plockats isär. Det enda som blivit kvar är själva Lego-bitarna som man så småningom samlat i en kartong för att kunna återanvända dem senare.

Vill man lösa ett komplext problem, t.ex. bygga ett hus eller en bil, bryter man ned det i ett antal mindre problem som är enklare att lösa. Sedan sätter man ihop de små enkla lösningarna till den stora komplexa lösningen. Principen heter *modularisering* och kan användas vid nästan all problemlösning. Ett stort komplext problem bryts ned i mindre *moduler* – motsvarande Lego-bitarna – och bearbetas en i taget. I programmering är dessa moduler ingenting annat än *funktioner*. Stora program bryts ned i ett antal funktioner. Varje funktion löser ett visst delproblem som är oberoende av andra och antagligen enklare att koda än det stora programmet. Sedan gäller det att sätta ihop modulerna till det stora programmet.

För att kunna göra det måste varje modul kommunicera med sin omgivning. Även här kan man lära av Lego: Varje Lego-bit är konstruerad så att den passar in i en annan Lego-bit. De delar av Lego-biten som tillåter denna passning, kan anses som Lego-bitens *gränssnitt* mot andra Lego-bitar. På samma sätt har en funktion ett gränssnitt mot andra funktioner för att kunna kommunicera med dem. Även detta gränssnitt har två delar: För det första funktionens *parametrar* som importerar värden från omgivningen och för det andra funktionens *returvärde* som exporterar ett värde till omgivningen. Men sedan måste Lego-bitarna ”sättas ihop” vilket i programmeringstermer innebär att *anropa* den ena från den andra. Ett *anrop* av en funktion innebär att *aktivera* funktionen. Detta sker genom att ev. skicka till den parametrar, utföra koden som står i funktionen och ev. få tillbaka returvär-

det. Generellt finns det i ett program flera funktioner som anropar varandra. Det enklast tänkbara exemplet är att **`main()`** anropar en **`Funktion`** dvs **`main()`** är den *anropande* och **`Funktion`** den *anropade* funktionen. Då kan programflödet mellan dem se ut så här:



Återanvändning av kod

är det andra svaret på frågan varför man i programmering sysslar med funktioner. Samma idé finns bakom Lego-biten som minsta återanvändbara modul för att bygga i princip vad som helst. Har man i ett program löst ett litet delproblem som även dyker upp i andra sammanhang och vars kod kan vara relevant i andra program, så vill man ju helst inte satsa tid och resurser för att koda det en gång till. Man vill undvika att återuppfinna hjulet. Detta är inte bara av teoretiskt-estetiskt intresse utan även av stort ekonomiskt intresse. Det man gör är att lösa koden för det lilla delproblemet från det aktuella programmet och skriva den som en funktion för att kunna återanvända koden i vilket annat program som helst. Man behöver då endast anropa den från andra program. Det kräver förstås att den ursprungliga koden som kanske var skräddarsydd för just det speciella programmet då, nu som funktion måste formuleras på ett mer generellt sätt och förses med möjligheten att kunna kommunicera med andra program. Därför måste koden kompletteras med parametrar och returvärdet. Hela tanken bakom standardbibliotek – inte bara i C++ utan i alla programspråk – bygger på idén om återanvändning av kod. Även om man väljer att inte skriva egna funktioner kan man i alla fall inte komma ifrån att använda redan fördefinierade funktioner från standardbiblioteket.

Strukturering av program

är det tredje svaret på frågan varför funktioner, närmare bestämt egendefinierade funktioner, används i programmering. Genom att modularisera ett komplext problem som ska lösas med hjälp av datorn underlättar man inte bara själva lösningen (innehållet) utan kan även lättare få en strukturering av programkoden (formen). Det enklast tänkbara sättet att strukturera vilket program som helst är t.ex. att dela in det i ***inmatning – bearbetning – utmatning*** (sid 64). Dessa tre delar kan skrivas i var sin funktion vilka sedan anropas av **`main()`**. Denna huvudfunktion kan då bestå av ett få antal satser som endast anropar programmets olika funktioner. På så sätt har man från **`main()`** en övergripande kontroll över hela programflödet. Dessutom kan funktionerna lagras i separata filer och inkluderas med **`#include`**-direktiv i den fil som innehåller **`main()`**. Så kan man så småningom bygga upp sitt eget bibliotek av egendefinierade funktioner.

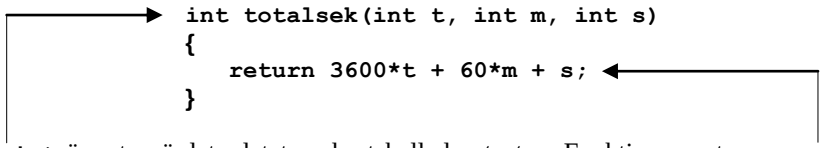
7.3 Definition och anrop av funktioner

Vi ska nu i detalj gå igenom alla steg hos funktioner *med* returvärde. I avsnitt ***7.6 Funktioner utan returvärde*** kommer den andra typen av funktioner, s.k. **`void`**-funktioner, att behandlas (sid 144).

Definition av en funktion med returvärde

```
returtyp Funktionsnamn(datatyp fpar1, datatyp fpar2, ...)
{
    sats(er);
    return uttryck;
}
```

Med returtyp menas datatypen till *returvärdet* och fpar står för *formell parameter*. Så kallas parametrar som förekommer i funktionens definition. Första raden är *funktionshuvudet* inklusive parentesen **`(`** ... **`)`** som innehåller listan över alla parametrar, därför kallad *parameterlistan*. Den kan innehålla en eller flera parametrar, men kan även vara tom. Oavsett antalet parametrar inkluderar man alltid, när man i beskrivande text nämner en funktion, parentesen **`()`** och lägger den till funktionsnamnet. Det gör man generellt för att skilja mellan funktioner och variabler. Parentesen är alltså kännetecknet för en funktion. Observera att funktionshuvudet inte avslutas med semikolon. Det är inte en sats utan bara huvudet till en funktion vars kropp följer. I programexemplet **`MyFirstFct`** är funktionen **`totalsek()`** definierad på följande sätt:



Detta **`int`** är returvärdets datatyp, kort kallad *returtyp*. Funktionen returnerar ett heltal. Men varför står returtypen framför funktionsnamnet? Det verkar – om man för ett ögonblick bortser från parameterlistan – som om **`int totalsek`** vore en deklarqtion av ”variabeln” **`totalsek`** till datatypen **`int`**. Denna tolkning är korrekt om man tar hänsyn till att **`totalsek`** är både funktionsnamn och returvärdets minnescell. Denna minnescell kan få sitt värde endast från **`return`**-satsen, när funktionen anropas. Medan returvärdet är funktionens output (utdata) är parametrarna funktionens input (indata). **`totalsek()`** har tre parametrar av typ **`int`** som vi döpt till **`t, m, s`**. De är definierade i parameterlistan: **`(int t, int m, int s)`**. Observera att det inte går att skriva **`(int t, m, s)`** som man kan göra vid definition av vanliga variabler. Visserligen är parametrar också variabler, men när de definieras i parameterlistan, måste de upprepa sina datatyper även om de är av samma typ. Det är den enda syntaktiska skillnaden mellan parametrar och vanliga variabler. I definitionen heter de *formella* parametrar därför att de definieras i parameterlistan som ”tomma” minnesceller i väntan på att bli fyllda med värden när funktionen an-

ropas. Deras namn saknar betydelse – bara man använder konsekvent samma namn i funktionskroppen. De *initieras* dvs tilldelas värden *första gången* när funktionen anropas. Vid anropet *importerar* **t, m, s** de erhållna värdena till funktionen dvs vidarebefordrar dem endast från **main()** till funktionen där de bearbetas. Returvärdet *exporterar* sedan resultatet ur funktionen till **main()**.

I *kroppen* till en funktion kan ett antal satser stå avgränsade med klammerparet **{ … }**. Klamrarnas uppgift är att gruppera satserna under funktionshuvudet till ett block. I funktionen **totalsek()** består detta block av en enda sats som kallas **return**-*satsen*. Denna sats returnerar uttryckets värde till funktionsnamnet. Men **return**-satsen gör en sak till: Den avslutar även funktionen. Eventuell kod efter den kommer att inte utföras. Därför ska **return**-satsen alltid vara funktionens sista sats. Den logiska slutsatsen är att det får finnas endast en **return**-sats i en funktion. Då **return**-satsen returnerar uttryckets värde till funktionsnamnet, måste namnet ha beredskapen at ta emot det. Detta innebär att **totalsek** samtidigt som det är funktionsnamnet, också är en variabel av typ **int** – med den begränsningen att den endast kan få sitt värde från **return**-satsen. **Totalsek** kan inte lagra returvärdet som är ett haltal, om det inte är via returtypen definierat till datatypen **int**.

Anrop av en funktion med returvärde

Funktionens definition är endast är en *mall*, en *föreskrift* om vad som *skulle* hända om funktionen anropas, jämförbar med ett matrecept som man skriver ned och stoppar i kökskåpets låda i väntan på att någon gång ta fram det och laga mat. Först när beredskapen till matlagning finns – alla ingredienser är handlade och finns på plats – kan matreceptet komma till användning som en algoritm. Samma sak är det med funktionens definition: den är endast en potentiell eller formell kod. Aktuell blir den först när vi anropar funktionen. Då börjar saker och ting att hända. I programmet **MyFirstFct** anropas funktionen **totalsek()** från funktionen **main()** i följande sats:

```
cout << … << totalsek(tim, min, sek) << … ;
```

Själva anropet består av den gråmarkerade koden dvs av att kalla funktionen vid namn och i parameterlistan skriva *lika många* parametrar som definitionen föreskriver – i det här fallet tre. Parametrar som förekommer i funktionens anrop – här **tim**, **min**, **sek** – kallas *aktuella*. Antalet aktuella parametrar *måste* vara lika med antalet formella parametrar – de som förekommer i funktionens definition. Annars blir det kompileringsfel. Samtidigt *borde* de aktuella parametrarna ha samma datatyp som de formella. Annars försöker kompilatorn att göra automatisk typkonvertering till måldatatypen. Dessutom måste vi se till att parametrarnas *ordning* stämmer dvs vi måste kontrollera om vi verkligen vill skicka **tim** till **t**, **min** till **m** och **sek** till **s** för exakt i den ordning vi skriver dem i anropet, kommer deras värden att överföras till de formella parametrarna i funktionens definition. Vi har döpt dem till **tim**, **min**, **sek**, definierat dem i **main()** som **int**-variabler och tilldelat dem värden via inläsning med **cin**-satsen strax före anropet.

Att anropet kan läggas i **`cout`**-satsen ovan beror på att **`totalsek()`** är en funktion *med returvärde* – dvs har en **`return`**-sats – och funktionsnamnet därmed samtidigt bär returvärdet i sig. I exemplet bakar vi in anropet i **`cout`**-satsen för att skriva ut returvärdet direkt på skärmen. Andra alternativ är att lägga anropet i en tilldelningssats till höger om tilldelningstecknet och låta en variabel ta hand om returvärdet eller anropa funktionen genom att kalla på namnet i en sats utan att ta hand om returvärdet:

```
cout <<    funktionsnamn(apar1, apar2, ...);
```

eller

```
variabel = funktionsnamn(apar1, apar2, ...);
```

eller

```
funktionsnamn(apar1, apar2, ...);
```

där **`apar`** står för aktuell parameter och **`funktionsnamn`** är den anropade funktionen. Självklart måste **`variabel`** och även alla aktuella parametrar vara definierade *före* anropet på vanligt sätt i den anropande funktionen – i vårt exempel i **`main()`**.

7.4 Deklaration av funktioner

I vårt första program om funktioner i förra avsnitt placerade vi den funktionen **totalsek** *före* **main()** dvs funktionens definition kom *före* anropet i **main()**. Detta behöver inte alltid vara så med tanke på att en funktion ska helst vara en separat modul som kan användas i andra program. Man kan i C++ placera funktionen även *efter* **main()** som en led i processen att isolera funktionen helt och hållet och placera den t.o.m. i en separat fil som en externlagrad funktion. För att kunna göra det behöver vi begreppet *deklaration av en funktion* som introduceras i följande exempel. Observera den lilla, men avgörande skillnaden mellan *definition* och *deklaration* av en funktion!

```
// Random.cpp
// Gör samma sak som programmet NestedFor
// In- och utmatning i tabellform görs i main()
// Slumptalsgenerering har flyttats till en funktion
// Funktionen deklareras i main() och definieras efter main()
#include <iostream>
#include <ctime>                // Krävs för funktionen time()
using namespace std;
/*************************************************************/
int main()
{
   int myRand(int, int);        // Deklaration av funktion

   srand(time(0));              // Skapar variation i slumpen
   int r, k, rader, kolumner;
   cout << "\nAnge antal rader och kolumner (t.ex. 20 15): ";
   cin >> rader >> kolumner;
   cout << "\nDet blir "<< rader*kolumner <<
                                      " tärningskast:\n\n";
   for (r=1; r<=rader; r++)
   {
       for (k=1; k<=kolumner; k++)
           cout << myRand(1, 6) << "   ";// Anrop av funktion
       cout << '\n';
   }
   cout << '\n';
}
/*************************************************************/
int myRand(int a, int b)         // Definition av funktion som
{                                // slumpar fram heltal i in-
   if (a < b)                    // tervall mellan a och b
      return a + rand() % (b-a+1);
   else
      return b + rand() % (a-b+1);
}
/*************************************************************/
```

Strax i början av **main()** står deklarationen till funktionen **myRand()**:

```
int myRand(int, int);
```

myRand() anropas i **main()** i varje varv av **for**-satsen medan definitionen av **myRand()** står efter **main()**. Om vi inte definierar funktionen i början stöter kompilatorn som går igenom koden rad för rad, *först* på funktionsanropet i **main()** *innan* den hittar funktionsdefinitionen. I så fall kan programmet inte kompileras för att kompilatorn inte kan tolka funktionsanropet. Deklarationen är i själva verket en förhandsinformation till kompilatorn som talar om att det kommer att anropas en funktion:

1. som heter **myRand()**
2. som returnerar ett värde av typen **int**
3. som har 2 parametrar
4. vars parametrar båda är av typen **int**

Denna information är både tillräcklig och nödvändig för att kunna kompilera funktionsanropen i programmet **Random**. Vad den talar om är att så här ser funktionen ut som jag kommer att anropa. Vad anropet innebär – själva koden (innehållet) som ska exekveras – kommer också att definieras senare. Pga deklarationens roll som en förhandsinformation pratar man ibland på engelska om *forward declaration*. I litteraturen finns även beteckningen *prototyp* av en funktion.

Signaturen

Deklarationens ovannämnda egenskaper 1, 3 och 4 – funktionens namn, antalet parametrar och parametrarnas datatyper, kallas funktionens *signatur* dvs dess igenkänningstecken. T.ex. är

```
myRand(int a, int b)
```

signaturen till en funktion som heter **slumpa** och har två parametrar av typ **int**. Funktioner med samma signatur anses vara *identiska*. Funktioner som skiljer sig på någon av signaturpunkterna 1, 3 eller 4, anses vara *olika*. Anmärkningsvärt är att returtypen inte ingår i signaturen, däremot funktionens namn samt parameterlista. Signaturen är alltså en del av deklarationen.

Hur deklarerar man funktioner?

Att deklarera en funktion – eller att skriva en prototyp – är väldigt enkelt: Man kopierar funktionshuvudet från definitionen och avslutar den med semikolon så att det uppstår en sats. Semikolonet är den avgörande skillnaden till funktionshuvudet, visar att deklarationen är en sats som måste avslutas med semikolon, medan funktionshuvudet är en rubrik som inleder funktionens definition. I så fall borde i exemplet ovan deklarationen skrivas så här: **int myRand(int a, int b);** Om man vill kan man göra det. Men man kan också utelämna parametrarnas namn **b** och **m** vilket vi har gjort i **Random**. Detta för att öka programmets läslighet genom att bättre kunna skilja en deklaration från ett funktionshuvud. Semikolonet kan lätt förbises vid hastig läsning. Däremot är det mer iögonfallande att parameternamnen fattas vilket indikerar att man har att göra med en deklaration. För kompilatorn gör

det ingen skillnad, den ignorerar alla parameternamn i deklarationen. Det är anmärkningsvärt att det inte ens blir kompileringsfel om man väljer helt andra namn på parametrarna i deklarationen än senare i funktionens anrop. Redan för att undvika godtyckligheten vid val av namn borde det vara rimligt att utelämna parametrarnas namn i deklarationen av en funktion. Huvudskälet för oss är dock bättre läslighet. Därför skriver vi i exemplet ovan deklarationen så här:

```
int myRand(int, int);
```

Deklarationen kan placeras på två ställen: antingen *före* **`main()`** eller *i* **`main()`** som vi gjort. En deklarationssats *före* **`main()`** alltså på samma plats som **`include`**-direktiven, skulle gälla inte bara i **`main()`** utan även i alla funktioner som ev. följer. Däremot en deklarationsats *i* **`main()`** gäller endast i **`main()`**. Med deklarationens giltighet syftar vi åt den plats i koden därifrån man anropar funktionen. I **`Random`** är det **`main()`**. Därför har deklarationssatsen placerats *i* **`main()`**.

Vad gäller själva slumptalsgenereringen i funktionen **`myRand()`** hänvisas till sid 117 där hantering av slumptal behandlas.

Nu kan vi visa ett körexempel på **`Random`** som samtidigt är ett körexempel på **`NestedFor`** då programmen ger ”samma” resultat, det första *med* och det andra *utan* funktioner. ”Samma” förstås om man bortser från att varje körning ger andra slumptalsvärden:

```
Ange antal rader och kolumner (t.ex. 20 15): 20 15

Det blir 300 tärningskast:

5   6   6   6   3   6   3   3   3   6   3   5   6   1   2
4   4   6   1   6   4   3   2   4   3   1   3   1   1   6
1   2   2   5   2   5   5   3   1   2   4   4   3   1   3
2   2   1   4   4   3   6   2   5   5   3   6   5   3   2
1   3   3   5   6   1   6   3   5   1   4   3   6   2   2
4   2   2   3   5   5   4   6   5   3   1   3   4   5   4
3   5   5   4   6   3   6   1   6   1   6   4   6   6   5
3   6   6   3   4   4   4   3   6   4   3   6   5   6   4
2   2   4   6   5   6   6   6   1   5   4   2   6   1   3
2   6   6   4   5   3   6   6   2   3   6   6   6   4   6
5   1   5   2   5   3   4   1   1   3   2   4   1   5   1
1   4   4   6   6   6   2   5   1   1   2   3   5   4   3
6   3   1   1   5   3   4   5   2   6   3   2   4   2   5
1   4   2   5   6   6   3   5   5   1   3   1   2   4   6
3   4   4   3   6   3   4   4   6   6   3   4   2   5   4
6   4   3   6   3   1   4   5   1   2   6   6   6   3   5
4   3   4   5   3   4   1   2   5   1   3   3   4   3   4
1   5   1   6   4   5   2   2   6   4   2   3   2   4   5
4   3   2   3   1   5   6   5   6   6   6   5   1   2   4
4   1   1   3   3   5   5   3   2   3   2   4   6   2   3
```

7.5 Externlagrade funktioner

Prototypen som behandlades i förra avsnitt var en led i processen att isolera en funktion och placera den i en separat fil som en *externlagrad* funktion. Man vill bygga separata moduler som kan användas även av andra program. Så länge en funktion är inbunden i samma fil som det program som anropar funktionen kan den inte användas av andra program. Man vill bryta ned program i oberoende moduler för att kunna dra nytta av återanvändning av kod (Modularisering sid 133). Detta leder till att man löser den anropade funktionen helt och hållet från den fil som innehåller den anropande funktionen, här **`main()`**, och lagrar den i en annan fil. Sedan måste modulerna länkas ihop. Ett exempel på detta finns i följande program:

```
// ExternFct.cpp
// Skriver ut en tabell över nettobelopp och ingående moms
// Programmet består av två funktioner:
// 1) main() matar in och ut samt anropar funktionen netto()
// 2) netto() som är externlagrad beräknar nettobeloppet
#include <iostream>
#include <iomanip>           // Krävs för setprecision()
using namespace std;

#include "Netto.h"           // Innehåller funktionen netto()

int main()
{
   float first, last, step, vatRate, brutto;
   cout << "\nTabell över brutto- och nettobelopp samt "
        << "ingående moms:\n";
   cout << "\n\tBruttobelopp för början av tabellen:\t";
   cin >> first;
   cout << "\toch för slutet av tabellen:\t\t";
   cin >> last;
   cout << "Ange steget för momstabellen    : ";
   cin >> step;
   cout << "Aktuell momssats i % (t.ex.25)  : ";
   cin >> vatRate;
   cout << fixed << setprecision(2);  // Efter detta kommer
                    // alla cout-satser skriva ut tal med
                    // 2 fasta decimaler
   cout << "\nBrutto\t\tNetto\t\tIngående moms\n"
        << "-----------------------------------------\n";
   for (brutto=first; brutto<=last; brutto=brutto+step)
                                    // 2 anrop av netto():
       cout << brutto << "\t\t" << netto(brutto, vatRate)
            << "\t\t" <<  brutto - netto(brutto, vatRate)
            << "  kr\n";
   cout << "\n";
}
```

`include`-direktivet **`#include "Netto.h"`**

som står strax före **main()** gör att kompilatorn hittar filen **Netto.h**, även kallad ***h****eaderfil*. Den innehåller definitionen av funktionen **netto()**. Filens ändelse **h** i de externlagrade funktionerna är inte obligatorisk utan endast en konvention som ska påminna om att det i dessa filer lagras funktioner som inte utgör kompletta program utan *delar* av fullständiga program. Syntaxen för inkludering av **h**-filer skiljer sig från syntaxen för biblioteksfilerna: Filens namn skrivs inom citationstecken för att skilja den från standardbiblioteksfiler. Kompilatorn ska inte söka den i den mapp som är avsedd för biblioteksfilerna utan i samma mapp som programfilen **ExternFct.cpp**. I detta fall måste vi placerat headerfilen i samma mapp som programfilen. Men det är även möjligt att placera den i en annan mapp och ange en korrekt sökväg i **include**-direktivet. Så här ser headerfilen ut:

```
// Netto.h
// Beräknar nettobeloppet utgående från ett bruttobelopp
// Definierar funktionen netto() som tar in ett bruttobelopp
// b samt momssatsen m och returnerar nettobeloppet
// netto() anropas från main() lagrad i filen ExternFct.cpp
// Inget fullständigt program, därför ändelsen h (headerfil)

float netto(float b, float m)        // Funktionsdefinition
{
   return 100*b/(100+m);             // Formeln för beräkning
}                                    // av nettobelopp
```

De formella parametrarna **b** och **m** kommer att ta emot sina värden från de aktuella parametrarna **brutto** och **vatRate** när **main()** anropar funktionen **netto()**. Funktionen beräknar sedan uttrycket i **return**-satsen och returnerar resultatet till **main()** som är placerad i programfilen **ExternFct.cpp**. Att funktionerna hittar varandra trots att de är placerade i olika filer, beror på att **h**-filen är kopplad till **cpp**-filen via **include**-direktivet. Programmet ovan består av *två* funktioner som anropar varandra: **netto()** och **main()** lagrade i två separata filer.

Vad programmet **ExternFct** i sin helhet gör är att läsa in önskade bruttobelopp för början och slutet på en tabell. Dessutom ska även ett steg anges som tillämpas i tabellen. Tabellen skriver ut givna bruttobelopp i en första kolumn och till varje bruttobelopp det beräknade nettobeloppet enligt en viss momssats som också läses in i början. Nettobeloppen hamnar i den andra kolumnen. Dessutom beräknas och skrivs ut i en tredje kolumn den moms som ingår i varje bruttobelopp. Huvudjobbet görs i en **for**-sats som i varje varv anropar funktionen **netto()** två gånger, första gång för att skriva ut nettobeloppet i den andra kolumnen, andra gång för att skriva ut den ingående momsen i den tredje kolumnen genom att subtrahera den från bruttobeloppet. Med tabulatorn **\t** justeras avstånden mellan kolumnerna. **\n** i slutet av **for**-satsens **cout**-sats åstadkommer radbyte i tabellen. **cout**-satsen direkt innan **for**-satsen skriver ut tabellhuvudet. Kvarstår hur vi får att alla värden i tabellen skrivs ut med två decimaler. Lösningen är:

```
cout << fixed << setprecision(2);
```

fixed är en manipulator som ändrar **cout**-strömmens inställningar så att alla decimaltal skrivs ut med ett fast antal decimaler, by default (automatiskt) med 6. **Setprecision(2)** upphäver denna defaultinställning och sätter antalet decimaler till 2. **setprecision()** som kräver inkludering av **iomanip**, sätter antal *siffror* utan **fixed** och antal *decimaler* med **fixed**. I körexemplet nedan har vi valt en momssats som är vanlig i Sverige. Men den generella formel som används i funktionen **netto()** tillåter även andra momssatser.

```
Tabell över brutto- och nettobelopp samt ingående moms:

        Bruttobelopp för början av tabellen:     125
        och för slutet av tabellen:              150
        Ange steget för momstabellen:            5
        Aktuell momssats i % (t.ex.25):          25

Brutto          Netto           Ingående moms
-----------------------------------------------
125.00          100.00          25.00  kr
130.00          104.00          26.00  kr
135.00          108.00          27.00  kr
140.00          112.00          28.00  kr
145.00          116.00          29.00  kr
150.00          120.00          30.00  kr
```

Nedan ges en matematisk härledning av formeln för nettobeloppet:

$$\text{netto} + \frac{\text{netto} \cdot \text{momssats}}{100} = \text{brutto}$$

$$\frac{100 \cdot \text{netto} + \text{netto} \cdot \text{momssats}}{100} = \text{brutto}$$

$$\text{netto} \cdot (100 + \text{momssats}) = 100 \cdot \text{brutto}$$

$$\text{netto} = \frac{100 \cdot \text{brutto}}{100 + \text{momssats}}$$

Denna formel används i funktionen **netto()** i satsen `return 100*b/(100+m);` Skulle man vilja skriva en funktion för ingående moms kan man sätta in resultatet ovan i sambandet *ingående moms = brutto – netto* och använda följande direkt formel (lämpligt som övningsuppgift):

$$\text{ingående moms} = \frac{\text{brutto} \cdot \text{momssats}}{100 + \text{momssats}}$$