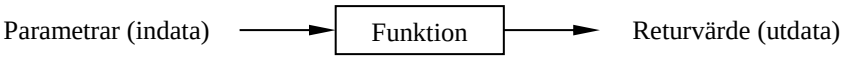
7.1 Funktionsbegreppet i programmering

Begreppet *funktion* härstammar från matematiken: Där har man en formel y = f(x) där f är ett uttryck som beräknar y utgående från x. Vi säger: y är en funktion av x. Denna matematiska syn på funktion har tagits över till programmering som ett underliggande koncept och som en historisk utgångspunkt. Men begreppet har fått inom programmering en bredare tolkning då den tillämpats på *all* datoriserad problemlösning. Visserligen kodar man i programmering också matematiska funktioner, men man inkluderar även kod som implementerar vilken annan funktionalitet som helst. Så blir funktionen en *del* av programmet – en logiskt sammanhängande *modul* som löser en specifik uppgift, men som även kan användas i andra program.

En funktion är kod som definieras som en namngiven modul och placeras utanför **`main()`**, före, efter eller externt.

Koden utförs inte förrän funktionen anropas i **`main()`**.

Vid anropet kan funktionen ta emot indata, s.k. parametrar, bearbeta dem och returnera utdata, ett s.k. returvärde.



I funktioner stoppar man indata och får ut utdata: Indata kallas även för *parametrar* (argument) och utdata för *returvärde*. En funktion kan ha inga, en eller flera parametrar. Däremot kan en funktion endast ha *ett* returvärde, vilket är ett arv från matematiken. I programmeringen finns även funktioner utan returvärde, som då kallas för **`void`**-funktioner. Funktionen *bearbetar* de inkommande parametrarna och returnerar returvärdet eller inget alls. Som separat och namngiven modul kan en funktion anropas i **`main()`** eller i andra funktioner resp. program. I denna bemärkelse är en funktion ett *underprogram*, på eng. *subroutine* eller *procedure*.

Man kan jämföra en funktion med en *”black” box*. *”Black”* är den så länge vi inte vet hur den fungerar ”inuti”, bl.a. om vi inte själva definierat funktionen. Det gäller t.ex. för de biblioteksfunktioner vi hittills använt i våra program: **`sizeof()`**, **`_getch()`**, **`rand()`**, **`srand()`** och **`time()`**. Dessa är *förprogrammerade* och lagras i bibliotek. Vi inkluderar dem i våra program för att dra nytta av deras funktionalitet, utan att behöva veta hur de exakt är kodade. Biblioteken i alla programmeringsspråk består av sådana ”svarta” lådor. Fr.o.m. detta kapitel kommer vi att skriva egna funktioner som fortfarande är lådor, men inte längre svarta.

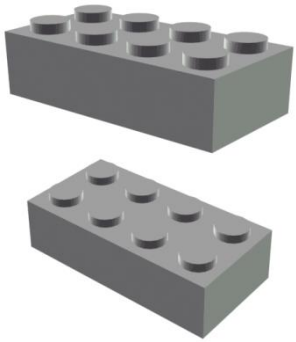
Den enda funktion som vi hittills definierat själva – och det har vi gjort i *alla* våra program – är **`main()`**. Den är unik därför att den är programmets exekveringspunkt. I alla procedurala programspråk bildar funktioner programmets byggstenar (moduler). Att C++ som är objektorienterat, även bygger på funktioner och inte ba-

ra på klasser, beror på språkets rötter i C som är proceduralt, men inte objektorienterat. *C#* och *Java* t.ex. bygger endast på klasser som inkapslar sina funktioner i klasser och döper dem till *metoder*. I *Python* har man återvänt till C/C++ modellen och tillåter fristående funktioner vid sidan av metoder.

Varför funktioner?

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. Man delar upp det stora programmet i mindre, logiskt meningsfulla moduler (delproblem), för att kunna klara av komplexiteten. Frågan, varför man i programmering sysslar med funktioner, har flera svar. Det första är:

1. Modularisering eller Lego-principen

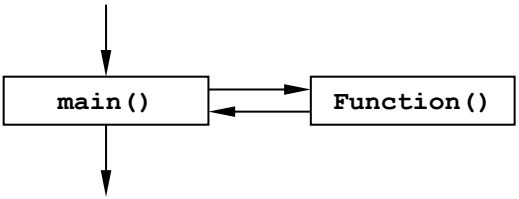


De flesta har väl någon gång som barn, eller tillsammans med sina barn, byggt ett Lego-hus, en Lego-bil eller liknande. Efter ett tag har huset kanske rasat och nya tekniska underverk har konstruerats. Men även de har någon gång plockats isär. Det enda som blivit kvar är själva Legobitarna (modulerna) som man 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. Varje modul löser ett delproblem som är oberoende av andra, är mindre än det stora problemet och därmed enklare att lösa. Sedan gäller det att sätta ihop modulerna till den stora lösningen. I programmering är dessa moduler *funktioner*.

För att att sätta ihop det hela 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 *an-*

rop av en funktion innebär att *aktivera* funktionen. Detta sker genom att ev. skicka ujtill den parametrar, utföra koden som står i funktionen och ev. få tillbaka returvärdet. Generellt finns det i ett program flera funktioner som anropar varandra. Det enklast tänkbara exemplet är att **`main()`** anropar en **`Function()`** dvs **`main()`** är den *anropande* och **`Function()`** den *anropade* funktionen. Då kan programflödet mellan dem se ut så här:



2. 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. Man löser koden för det lilla delproblemet från det aktuella programmet och skriver den som en funktion för att kunna återanvända koden i vilket annat program som helst. Det kräver att den ursprungliga koden som kanske var skräddarsydd för just det speciella programmet då, nu måste formuleras om så att den kan kommunicera med andra program. Dvs koden måste kompletteras med parametrar och ev. 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.

3. 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*. 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 bygga upp sitt eget bibliotek av egendefinierade funktioner.

Utdrag ur boken Progammering 1 med C++