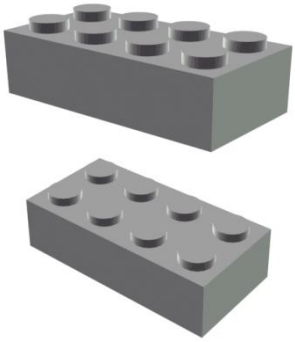
# 4.1 Funktionsbegreppet i programmering

Begreppet *funktion* härstammar från matematiken: Man har en formel y = f(x) som beräknar ett tal y utgående från ett annat tal x och 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 under tiden har begreppet vidareutvecklats och fått en bredare tolkning då den inom programmering tillämpats på all datoriserad problemlösning. I programmering inkluderar funktioner även matematiska problem, men är inte begränsade till dem.

## *Modularisering eller Lego-principen*

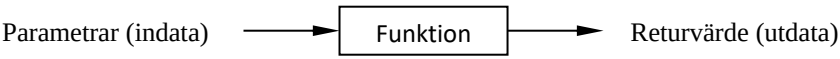


De flesta har väl någon gång som barn, eller tillsammans med sina barn, byggt ett hus, en bil eller liknande med Lego-bitar. Efter ett tag har huset kanske rasat och nya tekniska underverk har konstruerats. Men även de har någon gång plockats isär. Det enda som blivit kvar är själva Lego-bitarna som 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. 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*:

En funktion är kod som definieras som en namngiven modul.
Koden utförs inte förrän funktionen anropas.

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




Man kan jämföra en funktion med en ”svart” låda i vilken man stoppar indata och får ut utdata: Indata kallas även *parametrar* (argument) och utdata *returvärde*.

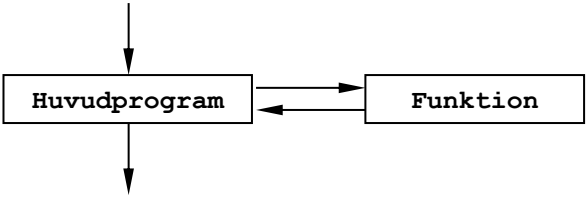
En funktion kan ha inga, en eller flera parametrar. Den kan ha inget eller endast *ett* returvärde, dvs en funktion kan inte ha flera returvärden, vilket är ett arv från matematiken. Både parametrarna och returvärdet kan vara tal, tecken, strängar, sanningsvärden, objekt eller andra datatyper. Funktionen bearbetar de inkommande parametrarna och returnerar returvärdet eller inget alls. I denna bemärkelse är en funktion ett *underprogram*, på eng. *subroutine* eller *procedure*.

”Svart” är lådan så länge vi inte vet hur den fungerar ”inuti”, dvs så länge vi inte själva definierat funktionen. I så fall använder vi den endast för att lösa ett visst problem. Det gäller t.ex. för de biblioteksfunktioner som vi hittills använt i våra program: **`document.writeline()`**, **`parseInt()`**, **`prompt()`**, **`alert()`** och andra. De är förprogrammerade och lagras i bibliotek. Vi anropar dem i våra program för att dra nytta av deras funktionalitet, utan att behöva veta hur de i detalj är konstruerade. Bibliotek som består av sådana ”svarta” lådor finns i alla programmeringsspråk.

## Gränssnitt

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. Därför har varje Lego-bit två uppsättningar av *taggar*, en på varje sida. Taggarna är de delar av Lego-biten som tillåter denna passning och kan anses som Lego-bitens *gränssnitt* mot andra Lego-bitar. På samma sätt har en funktion ett gränssnitt mot andra delar av koden, 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 returvärdet. Generellt finns det i ett program flera funktioner som anropar varandra. Det enklast tänkbara exemplet är att huvudprogrammet anropar en **`Funktion`** Då kan programflödet mellan dem se ut så här:




## Varför funktioner?

Kan man inte helt enkelt skriva kod rakt ned? 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. Lösningen är: man delar upp det stora programmet i mindre, logiskt meningsfulla delproblem, för att kunna klara av komplexiteten. Det finns i huvudsaken två viktiga skäl för användning av funktioner:

## *1. Återanvändning av kod*

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.

## *2. Strukturering av program*

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 huvudprogrammet en övergripande kontroll över hela programflödet. Så kan man så småningom bygga upp sitt eget bibliotek av egendefinierade funktioner.

Funktionen `max` (nedan) löser problemet att bestämma det största talet bland tre givna tal. Men detta problem kan även förekomma i andra sammanhang. Och då vill man helst använda den redan befintliga algoritmen som en återanvändbar modul, utan att behöva återuppfinna hjulet.

## *Vår första funktion*

```
1  <!-- MaxFct.html
2       Definierar och anropar funktionen max() som bestämmer
3       det största bland tre tal  -->
4  <title>En egendefinierad funktion</title>
5  <script>
6
7     function max(a, b, c)   // Definierar funktionen max()
8     {
9        tmp = a              // Antar att a är störst
10       if (b > tmp)
11          tmp = b           // Byter till b om b är större
12       if (c > tmp)
13          tmp = c           // Byter till c om c är större
14       return tmp           // Returnerar tmp till max()
15    }
16
17    no1 = parseInt(prompt('Mata in ett tal'))  // Inläsning
18    no2 = parseInt(prompt('Mata in ett tal till'))
19    no3 = parseInt(prompt('Mata in ett tredje tal'))

20    noMax = max(no1, no2, no3)       // Anropar funktionen max()

21    document.writeln('<h2>' + noMax + ' är det största talet ' +
22         'bland ' + no1 + ', ' + no2 + ' och ' + no3 + '.</h2>')
23 </script>
24
25 Ladda om sidan (Ctrl-R) för att köra om skriptet.
```

Raderna **7-15** definierar funktionen max() och rad **20**n anropar den.

Rad **7** kallas för funktionens *huvud* och inleds med det reserverade ordet **function** (sid 11). Funktionens *namn* är **max()**. Parentesen **(a, b, c)** kallas för *parameterlistan*. **a, b** och **c** är funktionens *formella parametrar*, medan **no1, no2** och **no3** som står i funktionsanropet (rad **20**), kallas för *aktuella parametrar*. Vid anropet kopieras de inlästa värdena från de aktuella till de formella parametrarna. På så sätt hamnar de i funktionen, där deras största värde bestäms.

Efter huvudet står funktionens *kropp* inom måsvingar (rad **8-15**). Kroppen avslutas med en s.k. **return**-sats som med hjälp av variabeln **tmp** returnerar det största värdet till namnet **max()**. På så sätt hamnar funktionens *returvärde* i programmet, när funktionen anropas på rad **20**. Eftersom namnet **max()** bär returvärdet måste anropet inbakas i en tilldelningssats, så att variabeln **noMax** kan ta emot detta värde som slutligen skrivs ut (rad **21**). Att funktionen **max()** innehåller en **return**-sats ger upphov till att kalla **max()** för en *funktion med returvärde*. Det finns i JavaScript även *funktioner utan returvärde*. Dessa saknar **return**-sats.