7.6 Funktioner utan returvärde

Hittills hade alla våra funktioner returvärden. Det var *en* typ av funktion, en ganska viktig sådan. Men funktioner med returvärde har en begränsning: De kan returnera endast *ett* värde, *ett* tal, *ett* tecken, *ett* sanningsvärde, *en* sträng eller *ett* objekt. Med *returvärde* menar vi alltid det som returneras av **return**-satsen via funktionsnamnet. En annan typ av funktioner är sådana som inte har något returvärde alls. I denna bemärkelse pratar vi nu om *funktioner utan returvärde* som även kallas **void**-funktioner. I några andra programmeringsspråk heter de *procedurer*.

> En **void**-funktion är en funktion som inte returnerar något värde. I funktionshuvudet ersätter **void** returtypen.
> En **void**-funktion har i regel ingen **return**-sats eller en tom **return**-sats: **return;** vilket är samma sak.

void är i C++ ett reserverat ord som kan tolkas som ”ingenting”. I vissa fall kan **void** ersätta datatypen. När den i funktionshuvudet ersätter datatypen till returvärdet och skrivs framför funktionsnamnet, eliminerar den returvärdet och definierar en **void**-funktion. T.ex. är **void main()** ett helt korrekt funktionshuvud som medför att **return**-satsen i **main()** antingen måste strykas eller ersättas av en tom **return**-sats. Även en tom **return**-sats avslutar funktionen var i kroppen den än skrivs vilket innebär att all kod efter den inte utförs. **void** kan även dyka upp i parameterlistan till en funktion och betyder då att funktionen inte har några parametrar. T.ex. är även **main(void)** ett korrekt funktionshuvud som är identiskt med **main()**. Däremot får **void** inte ersätta datatypen i deklarationer av vanliga variabler. Det strikt typbestämda språket C++ tillåter inte att en variabel definieras till ”ingen datatyp”. Följande är ett exempel på en **void**-funktion som definieras och lagras i en separat fil:

```
// Compare.h
// Funktion som jämför två tal på likhet, mindre än, större
// än och skriver ut resp. meddelande. Tom return-sats avslu-
// tar funktionen om talen är lika stora.
void compare(int g, int h)
{
   if (g == h)
   {
      cout << "\aGrattis, du har gissat rätt!\n\n";
      return;         // Tom return-sats avslutar funktionen
   }                  // utan att returnera något värde
   if (g < h)
      cout << "För LITET, försök igen !\n\n";
   else
      cout << "För STORT, försök igen !\n\n";
}
```

Meddelandetexterna tyder på att denna funktion används i samband med ett program som gissar tal. Det är faktiskt Gissa tal- spelet som vi tidigare behandlat i ett antal varianter: **GissaTal_1** (sid 101) och **GissaTal_2** (sid 106). Låt dem passera revy. Nu ska vi modularisera **GissaTal_2**. Vilka moduler – logiskt meningsfulla delproblem – finns i den? En av dem är den ovan definierade **void**-funktionen **compare()**. Där löses följande delproblem: Jämför det gissade talet **g** med programmets hemliga slumptal **h** och skriv ut ett grattismeddelande om de är lika. Dessutom meddela, om de inte är lika, mindre eller större. För att kunna lösa denna uppgift måste **compare()** ta in värdena till det gissade och hemliga talet via parametrarna **g** och **h**, jämföra dem med varandra och avsluta i fall att **g = h** dvs om användaren gissat rätt. Just i detta fall avslutar satsen **return;** som står i den första **if**-satsen, funktionen: Koden efter den utförs inte. En **return**-sats avslutar alltid en funktion. Utan den skulle efter grattismeddelandet även skrivas ut: **För STORT, försök igen!** vilket inte vore önskvärt. Den tomma **return**-satsen returnerar dock inget värde. Därför är **compare()** en **void**-funktion.

Den andra modulen i applikationen är:

```
// GuessVoid.cpp
// Gissa tal-spelet med slumptal i upprepad dialog
// Kan avslutas även om användaren inte gissar rätt
// Slumptal genereras i intervall med funktionen myRand()
// Gissningen testas i void-funktionen compare()
// Båda är externlagrade och inkluderas här
#include <iostream>
#include <ctime>
using namespace std;

#include "Compare.h"                    // Innehåller compare()
#include "..\MyRand.h"                  // Innehåller myRand()

int main()
{
   srand(time(0));
   int guessedNo, secretNo = myRand(1, 20); // Anrop myRand()

   do
   {
     cout << "\nGissa tal mellan 1 och 20 (Avsluta med 0): ";
     cin >> guessedNo;
     cout << "\n\t";
     if (guessedNo == 0)
     {
        cout <<"\nProgr.s hemliga tal var:\t" <<
                                          secretNo <<'\n';
        break;                          // Bryter do-loopen
     }
     compare(guessedNo, secretNo);      // Anrop av compare()
   } while (guessedNo != secretNo);
}
```

I denna modul finns **main()** som anropar **compare()** i en **do**-loop. Loopen håller igång dialogen för att användaren ska kunna fortsätta att gissa ett nytt tal i fall han/hon gissat fel. Loopen avslutas när avslutningsvillkoret **(guessedNo != secretNo)** är falskt dvs när **guessedNo** är lika med **secretNo**, när användaren gissat rätt. **if**-satsen i **do**-loopen ger användaren möjligheten att kunna avsluta och få reda på programmets hemliga tal genom att mata in **0**. Men det är inte loopens avslutningsvillkor som bryter loopen utan **break**-satsen. Den hoppar över resten av koden på ett liknande sätt som den tomma **return**-satsen i **compare()**.

En viktig praktisk konsekvens av **void**-funktioner är att anropet till skillnad från funktioner med returvärde inte behöver – ja inte får – inbäddas i en **cout**- eller tilldelningssats. **void**-funktioner anropas helt enkelt med namn och väl definierad parameterlista, om sådan finns. I vårt exempel finns två parametrar:

```
compare(guessedNo, secretNo);
```

Då själva funktionsnamnet inte längre bär något värde, kan det varken skrivas ut eller tilldelas någon variabel. Därför behöver vi inte längre någon variabel som tar hand om returvärdet. Det enda anropet gör är att exekvera den kod som står i funktionens definition, närmare bestämt i kroppen, efter att ha överfört parametrarnas värde till funktionen. Från funktionen kommer inget tillbaka. Den tredje modul som ingår i programmet **GissaVoid** är slumptalsgenereringen:

```
// MyRand.h

int myRand(int a, int b)          // Funktion som slumpar fram
{                                 // heltal i intervallet mel-
   if (a < b)                     // lan a och b, inkl. a, b
      return a + rand() % (b-a+1);
   else
      return b + rand() % (a-b+1);
}
```

Det är ett delproblem som är logiskt helt oberoende av resten av programmet och lämpar sig som en separat modul. Vi har i ett tidigare program (sid 138) redan använt den som funktion, fast i samma fil. Nu tar vi över den direkt därifrån och skriver den i filen **MyRand.h** som inkluderas i **GuessVoid** med **#include "..\MyRand.h"**. Anledningen till **..** är att vi placerat filen i mappen strax över den aktuella mappen för att använda den även till andra program: **..** betyder *mappen ovan*. I **main()** anropas funktionen **myRand()** före **do**-loopen då det vid varje körning bara behövs ett hemligt tal, till skillnad från det gissade talet. I anropet har vi valt det lilla intervallet **(1, 20)** för att förenkla gissningen. **else**-delen i funktionen **myRand()** ska göra användningen säkrare: Även om man skulle anropa funktionen med **myRand(20, 1)** skulle slumptal genereras mellan **1** och **20**. Till skillnad från **compare()** är **myRand()** ingen **void**-funktion utan returnerar det genererade slumptalet till **main()** där det tilldelas variabeln **secretNo**. Hela Gissa tal-spelet består nu av dessa tre funktioner: **main()**, **compare()** och **myRand()**. Ett körexempel skulle likna det på sid 107.

7.7 Lokala och globala variabler

```
// Global.cpp
// Skriver ut en tabell över tillkommande eller ingående moms
#include <iostream>
#include <iomanip>
using namespace std;

int vatType;                              // Global variabel
/*********************************************************************/
float vat(float a, float v)               // Definition av funk-
{                                         // tionen vat()
   if (vatType == 1)
      return a*v / 100;                   // Tillkommande moms
   else
      return a*v / (100 + v);             // Ingående moms
}
/*********************************************************************/
int main()
{
   float first, last, step, vatRate, amount;
   cout << "Vill du lägga till(1) eller dra av(0) moms? " <<
                                                  " (1/0): ";
   cin >> vatType;
   cout << "Ange ett belopp för början                  : ";
   cin >> first;
   cout << "och för slutet på en momstabell            : ";
   cin >> last;
   cout << "Ange steget för momstabellen               : ";
   cin >> step;
   cout << "Ange aktuell momssats i procent (t.ex. 25)  : ";
   cin >> vatRate;
   cout << fixed << setprecision(2) << '\n';
   if (vatType == 1)
      cout << "Netto\t\tBrutto\t\tTillkom. moms\n";
   else
      cout << "Brutto\t\tNetto\t\tIngående moms\n";
   cout << "-----------------------------------------\n";
   for (amount=first; amount<=last; amount=amount+step)
   {
       cout << amount << "\t\t";
       if (vatType == 1)                          // Anrop av
          cout << amount  + vat(amount, vatRate)  // vat():
               << "\t\t" << vat(amount, vatRate) << " kr\n";
       else
          cout << amount  - vat(amount, vatRate)
               << "\t\t" << vat(amount, vatRate) << "  kr\n";
   }
   cout << "\n";
}
```

Programmet ovan består av de två funktionerna **vat()** och **main()**. Dessutom finns nu för första gången en variabel definierad utanför **main()** och även utanför funktionen **vat()**: Variabeln **vatType** är en s.k. *global* variabel, just därför att den är definierad både utanför **vat()** och **main()**. Anledningen för denna placering är att den används i båda funktionerna och måsta gälla i båda. För att kunna göra det måste den definieras globalt. Man kan även säga att variabeln **vatType** är *global* därför att den är definierad ovanför alla *block* i programmet.

Blockstruktur

Om *block* sa vi på sid 96:

> ” I C++ kallas ett antal satser som omsluts av klamrarna **{** och **}** för *block*. … Klamrarna är *gränser* mellan programmets olika delar. De sätter gräns för variablers räckvidd. För att överskrida dem måste vissa regler om *blockstruktur* beaktas.”

Dessa regler är inget annat än reglerna för lokala och globala variabler. Ett mycket typiskt exempel på block är funktioner. Deras kroppar bildar block när de anropas. Därmed ger ett program som innehåller funktionsanrop, upphov till blockstruktur. Programmet **Global** innehåller flera anrop av funktionen **vat()**. Ett av dem – det första – ger upphov till följande blockstruktur:

```
int vatType;                    // Global variabel

int main()
{
                                // Lokala variabler i main():
   float first, last, step, vatRate, amount;
   .
   .
   .
   if (vatType == 1)            // Funktionsanrop:
      cout <<  . . .  << vat(amount, vatRate) << . . .

                  float vat(float a, float v)
                  {     // Lokala variabler i vat()

                     if (vatType == 1)
                        return a*v / 100;
                     else
                        return a*v / (100 + v);
                  }
   .
   .
   .
}
```

Att blockstrukturen ritas så här, beror på att funktionen **vat()** anropas i **main()** vilket innebär att koden till **vat()** exekveras där anropet inträffar. Detta skapar blockstrukturen som följer händelseförloppet vid exekvering och inte ordningen funktionerna skrivits i koden. Se även blockstrukturen på sid 148.

Frågan som dyker upp är: Vad händer med en variabel när man överskrider blockgränserna? Dvs hur långt räcker en variabeldefinition? Man pratar om variablers giltighetsområde eller *räckvidd*, på engelska *scope*. Generellt gäller:

Regeln för räckvidden av variabler:

En variabels räckvidd börjar vid deklararationssatsen och slutar vid gränsen till det block där den deklarerats.

Globala variabler

Om vi tillämpar regeln ovan på t.ex. variabeln **momstyp** i programmet **Global**, visar blockstrukturen på förra sida att det inte finns något block som variabeln är definierad i och därmed inte heller någon gräns. **vatType** är ju definierad utanför alla block. Därav följer att variabelns räckvidd börjar vid definitionen och slutar först när programet tar slut vilket innebär att **vatType** är en global variabel. Med andra ord, regeln kan även användas för att skilja mellan globala och lokala variabler. Avgörande för frågan om en variabel är global eller lokal, är alltså *platsen* där den definieras. Definieras en variabel utanför alla block blir den global. Definieras en variabel i ett block, blir den lokal i det blocket. En global variabels räckvidd är obegränsad. Därför gäller den globala variabeln **vatType** i hela programmet och därmed i programmets alla funktioner. Globala variabler tränger genom alla blockgränser och gäller i alla inre block, så länge de inte omdefinieras. Därför gäller **vatType** både i **main()** och **vat()**. En annan egenskap av globala variabler är att de initieras automatiskt till 0 om de är tal, till nolltecknet om de är tecken och till tom sträng om de är strängar. I vårt programexempel utnyttjas inte denna egenskap. Dock är den av betydelse då lokala variabler beter sig annorlunda: De måste initieras explicit i programmet.

Lokala variabler

I **main()** finns det fem variabler som är lokala: **first**, **last**, **step**, **vatRate**, **amount** och **vat**. De gäller endast i **main()**, men inte i **vat()**.

Hur är det med lokala variabler i funktionen **vat()**, finns några där? I kroppen är inga variabler definierade. Hade sådana funnits hade de varit lokala. De enda variabler som används i **vat()** är parametrarna **a** och **v** som är definierade i parameterlistan. Angående parametrar i en funktion gäller generellt följande:

Parametrarna i en funktion behandlas som lokala variabler i funktionen.

Därför gäller variablerna **b** och **m** endast i **vat()**, men inte i **main()**. Ett försök att använda dem i **main()** skulle ge kompileringsfel. Gör gärna testet! Kriteriet för om en variabel är global eller lokal, är *platsen* där den definieras. En parameter som definieras i funktionens parameterlista tillhör funktionens huvud och därmed funktionen. Trots detta är parametrarna inte i alla avseenden likställda de lokala variabler som definieras i funktionens kropp. Därför säger vi inte att de *är* lokala variabler utan att de *behandlas som* sådana. Deras särställning i parameterlistan innebär att de till skillnad från kroppens lokala variabler, kan *kommunicera* med omgivningen dvs med andra funktioner. Kom ihåg att vi har kallat dem *formella parametrar* därför att de initieras (får sina värden första gången) när funktionen anropas. Dvs de tar emot sina värden från de aktuella parametrarna **amount** och **vatRate** vilka är definierade i **main()**. Värdena kopieras från **amount** till **a** och från **vatRate** till **v**. Nu kan vi se att den här kopieringen går över blockgränsen från lokala variabler i ett block till variabler som behandlas som lokala (parametrar), i ett annat block. Detta ger upphov till formuleringen: parametrarna bildar en del av funktionens *gränssnitt* mot omgivningen dvs mot andra funktioner. I själva verket är det funktionshuvudet i sin helhet som utgör gränssnittet.

Ett körexempel av programmet **Global** kan se ut så här:

```
Vill du lägga till(1) eller dra av(0) moms? (1/0): 1
Ange ett belopp för början                       : 66.50
och för slutet på en momstabell                  : 70
Ange steget för momstabellen                     : 0.50
Ange aktuell momssats i procent (t.ex. 25)       : 25

Netto           Brutto          Tillkom. moms
-------------------------------------------
66.50           83.12           16.62 kr
67.00           83.75           16.75 kr
67.50           84.38           16.88 kr
68.00           85.00           17.00 kr
68.50           85.62           17.12 kr
69.00           86.25           17.25 kr
69.50           86.88           17.38 kr
70.00           87.50           17.50 kr
```

Körexemplets indata har valts på så ett sätt som visar att programmet kan användas på väldigt olika och även udda sätt. Tabellsteget kan vara decimalt pga datatypen **float** till variabeln **step**. Man skulle kunna förfina tabellsteget ytterligare. Tabelländarna kan också vara decimala. Andra momssatser fungerar lika bra. Väljer man i början momstypen **0** dvs ingående moms och matar in annars samma indata som på sid 143 till programmet **ExternFct** får man exakt samma resultat.

Varför globala variabler?

Av vilken anledning har **vatType** definierats som global variabel i **Global**? För att förstå det måste vi gå tillbaka till programmets upplägg. Vilket problem ska

programmet lösa? Man ska kunna beräkna moms som tillkommer till ett nettobelopp eller moms som ingår i ett bruttobelopp. För att kunna behandla båda alternativ efterfrågas först användarens önskemål i början av **`main()`**:

```
Cout<< "Vill du lägga till(1) eller dra av(0) moms? (1/0): ";
cin >> vatType;
```

Svaret läses in och tilldelas **`int`**-variabeln **`vatType`**. Denna inläsning som enligt ledtexten görs med **`1`** eller **`0`** används i **`main()`** för att skilja mellan moms på nettobeloppet och momsdelen av bruttobeloppet för att skriva ut korrekt huvud till tabellen och beräkna rätt netto- resp. bruttobelopp. Men samma distinktion måste även göras i funktionen **`vat()`** vilket ger upphov till en variabel som är giltig i både **`main()`** och **`vat()`**, dvs en global variabel. Även i funktionen **`vat()`** måste **`vatType`** med en **`if`**-sats avgöra valet av rätt momsformel. Så, **`vatType`** är nödvändig i båda funktioner. Framför allt ska variabeln behålla sitt värde när man vid anropet går över från den ena till den andra dvs när man överskrider blockgränserna. Därför måste värdet vara lagrat i en och samma minnescell. Men generellt sett medför globala variabler nackdelar även om dessa inte är påtagliga i exemplet.

Användningen av globala variabler strider mot modularisering och återanvändning av kod. T.ex. kan funktionen **`vat()`** inte isoleras som en separat modul och användas i något annat program då den alltid är beroende av programmet **`Global`**. Orsaken är att den globala variabeln **`vatType`** svetsar samman dem. Vid kompilering av **`vat()`** som del av att annat program måste ju variabeln **`vatType`** vara definierad. Men den är inte definierad i **`vat()`**. Då skulle den inte gälla i **`main()`**. Ett annat skäl är att felsökning i större program blir svårare när man försöker spåra en global variabel genom många funktioner. Även kontrollen av programflödet och struktureringen av koden kan kompliceras. Därför:

Rekommendation för användning av globala variabler:

Var restriktiv i bruket av globala variabler och använd dem endast då det är absolut nödvändigt.

Egentligen borde bra programmering kunna undvika globala variabler. Därav följer frågan: Var det absolut nödvändigt att ha en global variabel i programmet **`Global`**? Vid närmare betraktelse (och lite mer kunskap) ser man nämligen att detta inte alls var absolut nödvändigt. Det finns följande alternativ till globala variabler:

Alternativet: Parametrisering av globala variabler

Detta innebär att förvandla den globala variabeln till en lokal variabel i **`main()`** och att ersätta den i funktionen med en ny parameter. Dvs man inför en ny, tredje parameter i **`vat()`** som får sitt värde överförd från **`main()`** vid funktionsanropet. Denna uppgift överlämnas till övningarna i slutet av detta kapitel, se övn 8.7.

7.8 Överskuggning av variabler

När vi pratade om blockstruktur ställde vi upp regeln för *räckvidden* av variabler (sid 149) som i sin tur gav upphov till *lokala* och *globala variabler*. Nu ska vi komplettera våra kunskaper ytterligare med ett koncept som löser namnkonflikter vilka kan uppstå när lokala och globala variabler har samma namn: *överskuggning* kallas det. På köpet kommer vi att lära känna den s.k. *räckviddsoperatorn*. Titta på följande program:

```
// Scope.cpp
// Globala och lokala variablers räckvidd (giltighetsområde)
// Överskuggning av variabler och räckviddsoperatorn ::
#include <iostream>
using namespace std;

int x;                          // Global variabel initieras
                                // automatiskt till 0
void local();
void update(int);               // Deklaration av funktioner

int main()
{
   int x = 5;                   // Lokal variabel i main()
   cout<<"Lokalt x i main() före fkt.anropen är "<< x<< '\n';
   local();
   update(10);
   local();
   update(10);
   cout<<"\nLokalt x i main() efter anropen är "<< x << '\n';
   cout<< "\nGlobalt x (hämtat med ::) är " << ::x << "\n\n";
   return 0;
}

void local()                    // Funktion som använder egen
{                               // lokal variabel
   int x = 55;                  // Lokalt x i local()
   cout << "\nLokalt x i local() är " << x
        << " i början av local()" << '\n';
   x++;
   cout << "\nLokalt x i local() är " << x
        << " vid slutet av local()" << '\n';
}

void update(int dx)             // Funktion som uppdaterar
{                               // global variabel
   cout << "\nGlobalt x är " << x << " i början av update()"
                                                  << '\n';
   x += dx;
   cout <<"\nGlobalt x är " << x << " vid slutet av update()"
                                                   << '\n';
}
```

Hur många variabler har vi i programmet **Scope**? Man ser bara variabeln **x**. Men är det hela vägen ett och samma **x** eller är det olika variabler med samma namn? Tillämpar vi våra kunskaper om lokala och globala variabler som vi lärt oss i förra avsnittet, kan vi konstatera följande: Det finns tre *olika* variabler som har samma namn **x**. De refererar till tre *olika* minnesceller dvs tre olika fysiska adresser med samma logiska namn i koden. Det kan de göra eftersom de definieras i olika block:

1. Först har vi där strax i början utanför **main()** den globala variabeln **x**. Platsen där den definieras, kallas *globalt namnutrymme*, en slags globalt ”block”. Därför gäller den i hela programmet och i programmets alla underblock dvs funktionerna **main()**, **local()** och **update()**. Det finns dock undantag, t.ex. vid överskuggning som kan förekomma i vissa underblock (se **2** och **3** nedan). Den globala variabeln **x** får i programmet följande värden:

Globalt x ~~0~~ ~~10~~ 20

Som global variabel initieras den automatiskt till **0** när den definieras. I det första anropet av funktionen **update()** i **main()** dvs i satsen **update(10);** ändras värdet till **10** och i det andra anropet till **20**, båda gånger genom satsen **x += dx;** Att det är den globala variabeln **x** som gäller i funktionen **update()**, beror på att den inte omdeklareras där. Till skillnad från funktionen **local()** skapas i **update()** ingen ny lokal variabel med samma namn. Därför är det **x** som används där, det globala **x**.

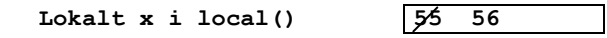
2. Sedan har vi en lokal variabel **x** i **main()** som endast gäller där. Nu uppstår uppenbarligen en konflikt: Denna lokala variabel har samma namn som den globala variabeln från punkt **1** som gäller i alla underblock och därmed även i **main()**. Båda kan inte gälla samtidigt, en måste slå ut den andra. Regeln säger följande:

> En lokal variabel i ett block överskuggar (slår ut temporärt) en global variabel med samma namn.

Överskuggning (eng. *overriding*) bör inte förväxlas med överskrivning (eng. *overwriting*). Överskriva kan man bara variabelns *värde* t.ex. med en ny tilldelning. Då blir det gamla värdet överskrivet för gott, kan aldrig återskapas och det nya värdet gäller i fortsättning. Överskuggning har inget att göra med variabelns värde utan med variabelns giltighet. I ett lokalt block kastar den lokala variabeln med samma namn temporärt en skugga över den globala variabeln. Bilden av skuggan har vi valt för att betona fenomenets temporära karaktär. Före och även efter det lokala blocket har den globala variabeln sin fulla giltighet. Så, i det lokala **main()**-blocket gäller den ”egna” lokala variabeln **x**. Den globala variabeln **x** gäller inte längre i **main()** utan är överskuggad av den lokala.

Den globala gäller endast före och efter **main()**. Hade vi valt ett annat namn för den lokala variabeln i **main()** hade förstås den globala haft full giltighet i **main()**. Men då hade vi inte lärt oss överskuggning, ett viktigt koncept inom programmering som tillämpas inte bara på variabler utan även på funktioner (se bokens sista kapitel om polymorfism). Den lokala variabeln **x** i **main()** initieras till **5** och ändras aldrig:

3. Slutligen används namnet **x** även för att definiera en lokal variabel **x** i funktionen **local()** på vilken samma överskuggningsregel (se förra sidan) tillämpas som på det lokala **x** i **main()**. Dvs även i funktionen **local()** överskuggar det lokala **x** som definieras där och initieras till **55**, programmets globala variabel **x** pga samma namn. Sedan ökas **x**-värdet med **1** genom **x++;** så att det blir **56** innan variabeln ”dör” då funktionen **local()** upphör:



Denna minnesbild uppstår i programmet **Scope** två gånger pga att funktionen **local()** anropas två gånger i **main()**.

Räckviddsoperatorn **::**

En annan nyhet i programmet **Scope** är räckviddsoperatorn som består av två kolon **::** utan mellanslag och kan sättas framför en variabel **::x** för att referera till det **x** som definierats i det globala namnutrymmet dvs här till den globala variabeln **x**. I **main()** används den i den sista utskriftssatsen för att hämta den globala variabelns aktuella värde. Ett körexempel av programmet **Scope** bekräftar detta:

```
Lokalt x i main() före fkt.anropen är 5
Lokalt x i local() är 55 i början av local()
Lokalt x i local() är 56 vid slutet av local()
Globalt x är 0 i början av update()
Globalt x är 10 vid slutet av update()
Lokalt x i local() är 55 i början av local()
Lokalt x i local() är 56 vid slutet av local()
Globalt x är 10 i början av update()
Globalt x är 20 vid slutet av update()
Lokalt x i main() efter anropen är 5
Globalt x (hämtat med ::) är 20
```