2 Grundläggande C++

C++-språket är en *utökning* av språket C. C++ är skapat av Bjarne Stroustrup på Bell laboratoriet i USA. Man kan se plustecknen som att de står för :

C		
+	=	En bättre och *säkrare* upplaga av C.
+	=	Verktyg för *objektorienterad* programmering.

Ex : Skriv ett program i C och i C++ som läser in radien för en cirkel och beräknar och skriver ut cirkelns area.

a) ANSI-C

```
/* Huvudprogram -- cirkel.c */

#include <stdio.h>
#define PI 3.14159

void main()
{
   float radie;

   /* Läs in radien */
   printf("Ge cirkelns radie : ");
   scanf("%g", &radie);

   /* Skriv ut arean */
   printf("Cirkelns area = %f\n", PI * radie * radie);

}
```

b) C++

```
// Huvudprogram -- cirkel.cpp

#include <iostream>
using namespace std;

void main()
{
   const float pi = 3.14159;
   float radie;

   // Läs in radien
   cout << "Ge cirkelns radie : ";
   cin >> radie;

   // Skriv ut arean
   cout << "Cirkelns area = " << pi * radie * radie << endl;
}
```

2.1 In- och utmatning

I C++ gör man in- och utmatningar med hjälp av fördefinierade objekt, så kallade *strömmar*. Dessa är definierade i inkluderingsfilen iostream.i namnrymden std. Där finns objekten :

cin som motsvarar stdin
cout som motsvarar stdout
cerr som motsvarar stderr(obuffrad)
clog som mosvarar stderr(buffrad).

Inmatning gör man med operatorn >> på objektet cin och utmatning med operatorn << på cout. Dessa operatorer betyder normalt höger- resp vänsterskift, men är i dessa fall omdefinierade att gälla in- resp utmatning. Mer om omdefiniering eller överlagring av operatorer senare.

Med de nya in- och utmatningsoperatorerna slipper man att speciellt ange ett format för hur variabeln ska tolkas, som i scanf("%g", &radie) utan detta sker *automatiskt utgående från datatypen*. I satsen cin >> radie så är det alltså typen på radie, som bestämmer hur de inmatade tecknen ska tolkas. Man slipper också att skicka adressen till variabeln som parameter eftersom man i C++ infört referensparametrar, se nedan.

Det finns funktioner (metoder) hos objektet cout för att formatera utmatningen när det exempelvis gäller antal positioner för utskrift och antalet decimaler :

```
// Skriv ut arean för cirkeln i 8 positioner(högerjusterat)
// med 2 decimaler

cout << "Arean = ";
cout.setf(ios::fixed, ios::floatfield);
cout.precision(2);
cout.width(8);
cout << pi * r * r << endl;
```

Anger man ingen formatering används *default-värden*. Exempelvis skrivs ett reellt tal ut vänsterjusterat i enklast möjliga format.

En noggrannare genomgång av strömmar, in- och utmatning kommer i kapitel 6.

2.2 Datatyper

I C++ finns samma enkla datatyper som i C. Bland de sammansatta typerna har datatypen *class* tillkommit. Typen class är en sammansatt datatyp, som används vid den objektorienterade programmeringen, för att kapsla in data och operationer till en abstrakt datatyp, se kapitel 3. De andra sammansatta typerna som exempelvis vektor, struct (post) etc är likadana som i C. När det gäller structar, unioner och egenuppräknade typer har man dock infört ett enklare sätt att definiera typen, utan att behöva använda typedef. När det gäller pekare och dynamisk allokering av minnesutrymme, har man också infört nya funktioner.

2.2.1 Pekare och dynamisk allokering

I C++ har man infört funktionerna *new* och *delete* för att dynamiskt allokera resp avallokera minnesutrymme. Dessa funktioner ersätter *malloc* resp *free* i C.

Ex : a) ANSI-C

```
#include <string.h>     /* strcpy         */
#include <stddef.h>     /* NULL           */
#include <stdlib.h>     /* malloc, free   */
#include <stdio.h>

void main()
{
   float *fp;
   char *str;


   /* allokera och avallokera ett flyttal */

   fp = (float *)malloc(sizeof(float));
   if ( fp != NULL)
   {
      *fp = 3.45;
      free(fp);
   }
   else
      printf("Kan ej allokera minne för flyttal!");


   /* allokera och avallokera en sträng med 50 tecken*/

   str = (char *)malloc(50*sizeof(char));
   if ( str != NULL )
   {
      strcpy(str, "Hello C++");
      free(str);
   }
   else
       printf("Kan ej allokera minne för sträng!");
}
```

b) C++

```
#include <cstring>      // strcpy
#include <cstddef>      // NULL
#include <iostream>
using namespace std;

void main()
{
   float *fp;
   char *str;


   // allokera och avallokera ett flyttal

   fp = new(nothrow) float;
   if ( fp != NULL)
   {
      *fp = 3.45;
      delete fp;
   }
   else
      cout << "Kan ej allokera minne för flyttal!";


   // allokera och avallokera en sträng med 50 tecken

   str = new(nothrow) char[50];
   if ( str != NULL )
   {
      strcpy(str, "Hello C++");
      delete [] str;    //OBS! Tom hakparentes för alla tecken
   }
   else
       cout << "Kan ej allokera minne för sträng!";
}
```

För att testa att man verkligen kunnat allokera upp det minne man eftersträvat kan man som ovan *testa på pekarvärdet efter allokering*. Är detta NULL eller 0 har man misslyckats. I C++ måste man skicka med nothrow till new-funktionen, om man vill testa på NULL, annars kastar new-funktionen en exception, som man måste ta hand om, se kap 7.

Har man många allokeringar i sitt program kan det bli väldigt många NULL-test. Man kan istället i C++ definiera en felhanteringsfunktion, som man sedan alltid hänvisar till då allokeringen misslyckades.

Ex : Skriv ett program som dynamiskt allokerar en vektor av heltal med inläst antal element. Finns ej utrymme ska en felhanteringsrutin anropas och programmet terminera. Finns det utrymme ska tresiffriga tal slumpas till vektorn.

```
#include <cstdlib>      // srand, rand
#include <ctime>        // time i srand-anropet
#include <iostream>
using namespace std;


// felhanteringsfunktion
void new_fel()
{
   cerr << "Kan ej allokera minne!" << endl;
   cerr << "Programmet avslutas! " << endl;
   exit(1);
}

// huvudprogram
void main()
{
   int nr, *ivec;

   // hänvisa alla allokeringsfel till funktionen new_fel
   set_new_handler(new_fel);

   // läs in aktuellt antal
   cout << "Ge antalet vektorelement : ";
   cin >> nr;

   // allokera
   ivec = new int[nr];
   // om du ej lyckades allokera anropas new_fel här

   // slumpa
   randomize();
   for ( int i = 0; i < nr; i++ )        // OBS! int i
      ivec[i] = 100 + random(900);

   // skriv ut
   for (int i = 0; i < nr; i++ )
   {
      if ( (i % 10) == 0 )
         cout << endl;
      cout.width(5);
      cout << ivec[i];
   }

   // avallokera
   delete [] ivec;
}
```

OBS! I C++ kan variabeldefinitioner göras överallt i koden. Detta bör dock tillämpas *sparsamt*. Ett exempel då det kan tillämpas är för loop-variabler som i exemplet ovan. Loop-variablerna kommer bara att vara definierade i loopen.

2.2.2 Structar, unioner och egenuppräknade

För att få riktiga typnamn för structar i C får man använda *typedef*. I C++ får man ett typnamn direkt med struct-definitionen.

Ex :

a) ANSI-C

```
struct bok
{
   char titel[40];
   char forfattare[30];
   struct bok *next;
};

struct bok min_bok;          /* variabel av typen struct bok */
struct bok *bokhylla;        /* pekare till variabel av typen
                                struct bok */
```

b) ANSI-C

```
typedef
struct bok
{
   char titel[40];
   char forfattare[30];
   struct bok *next;
} boktyp;

boktyp c_bok;                /* variabel av typen boktyp */
boktyp *bokhylla;            /* pekare till variabel av
                                typen boktyp */
```

c) C++

```
struct bok
{
   char titel[40];
   char forfattare[30];
   bok *next;
};

bok cplus_bok;               // variabel av typen bok
bok *bokhylla;               // pekare till variabel av typen bok
```

Ett sätt att spara minnesutrymme är att använda *unioner* istället för structar. I en union kan termerna vara av två eller flera typer *dock ej samtidigt*. Även här är syntaxen enklare i C++ än i C.

Ex : C++

```
union mix
{
   int hel;             // heltal eller
   float flyt;          // flyttal, totalt 4 byte
};

mix z;                  // variabel av typen mix

z.hel = 5;              // utrymmet tilldelas ett heltal
z.flyt = 6.7;           // utrymmet tilldelas ett flyttal
                        // som skriver över heltalet
```

Bitarna i en union kan alltså *tolkas* på de olika sätt som anges av unionens termer.

För egenuppräknade variabler räknar man upp de tillåtna värdena. Även här har det blivit enklare att definiera variabler i C++.

Ex: En egenuppräknad dagtyp

```
enum  dagtyp {mandag, tisdag, onsdag, torsdag, fredag, lordag,
              sondag};

dagtyp dag;       // varibeln dag definieras

dag = torsdag;    // dag tilldelas ett av de 7 tillåtna värdena
```

I senare versioner av C++ har man infört en egenuppräknad typ, bool, som har värdena false eller true. Logiska uttryck ska då vara av denna typ istället för av heltalstyp, som tidigare då 0 var falskt och allt annat var sant.

Datatypen bool är definierad enligt:

```
enum bool {false, true};
```

Ex: Logiska uttryck som

a) heltal

```
int ok = 0;

while (!ok)
{
```

b) bool

```
bool ok = false;

while (!ok)
{
```

2.2.3 Typkvalificerare

I C++ kan man som i C använda *typkvalificerare* för att definiera att en variabel ska ha ett *konstant* värde eller ett *flyktigt* värde. En variabel måste kvalificeras flyktig om den uppdateras via hårdvaran.

Ex : Istället för att använda macrot

```
#define PI 3.14159
```

för att definiera konstanter kan man istället använda typkvalificeraren const enligt :

```
const float pi = 3.14159;
```

vilket innebär att variabeln pi är av typen 'en konstant float' och dess värde kan ej ändras.

Ex : Läs och skriv tecken från en serieport på en processor vars uart är mappad till adressen 0xc0000000. Uartens statusregister finns på offset 1 till denna adress, där bit nr 0 sätts om det finns tecken att hämta och bit nr 5 sätts om det finns tecken att skicka. Tecknet som ska hämtas eller skickas finns på offset 3 till den angivna adressen. Använd

a) bitoperatorer

```
.....
volatile unsigned char *uart = (unsigned char *)0xc0000000;
const int sra = 1;           // offset till uartens statusregister
const int rxa = 3            // offset till uartens teckenregister
unsigned char ch;
while (!(uart[sra] & 1));    // vänta på klart att hämta (busy wait)
ch = uart[rxa];              // hämta tecken
.....
uart[rxa] = ch;              // tecken som ska sändas
while (!(uart[sra] & (1<<5)); // vänta på klart att sända (busy wait)
```

b) bitstruct

```
......

struct portstatus
{
   unsigned int rxready :1;   // bit nr 0
   unsigned int overrun :1;
   unsigned int parerr  :1;
   unsigned int framerr :1;
   unsigned int rxbreak :1;
   unsigned int txready :1;   // bit nr 5
   unsigned int txfifo  :1;
                        :1;   // bit nr 7 ej använd
};


volatile portstatus *ps = (portstatus *)(0xc0000000 + 1);


volatile unsigned char *data = (unsigned char *)(0xc0000000 + 3);


unsigned char ch;

while (!ps->rxready);         // vänta på klart att hämta (busy wait)
ch = *data;                   // hämta tecken

.....

*data = ch;                   // tecken som ska sändas
while (!ps->txready);         // vänta på klart att sända (busy wait)
```

OBS! Avreferens av pekare som i `*data` och av postpekare som i `ps->txready`.

Utan kvalificering med *volatile* skulle kompilatorn optimera koden så att uart-värdet blir inläst till ett processorregister *en gång* och därefter sker bitkontrollen mot processorregistret som aldrig ändras. *Volatile förhindrar alltså en sådan optimering* och uart-pekaren pekar alltid på den verkliga uarten.

2.2.4 Typkonvertering

Typkonvertering kan göras *implicit*, alltså automatiskt av kompilatorn, eller *explicit* dvs att man uttryckligen säger till om att konvertera till annan typ.

Implicit konvertering göres exempelvis i beräkningar av uttryck som innehåller olika typer. Alla typer konverteras till den högsta (största) för att ej förlora i noggrannhet :

Ex :

```
int a = 3;
float c, b = 4.5;

c = a + b;  // a konverteras till float innan additionen
```

Vid tilldelningar och initieringar sker också konvertering till den typ som finns på vänstersida. Här kan en högre typ omvandlas till en lägre och information eventuellt gå förlorad.

Ex :

```
// initiering
int a = 65.3;     // a = 65
char c;

// tilldelning
c = a;            // c = 'A' (ASCII-koden för 'A' är 65)
```

Implicit konvertering sker också vid anrop av funktioner där den aktuella parametern kan omvandlas till den formella parameterns typ. Detta kan jämföras med en initiering enl ovan.

Explicit konvertering kan du göra själv för variabler eller objekt om konverteringen finns definierad. I C++ kan man också konvertera med ett *funktionsliknande* anrop istället för att som i C skriva den typ, som man ska konvertera till, inom parentes.

Ex :

```
int a = 3, b = 2;
float c;

c = a / b;              // heltalsdivision, c = 1.0
c = (float)a / b; // flyttalsdivision, c = 1.5
c = float(a) / b;       // flyttalsdivision, c = 1.5 (C++)
```

2.3 Funktioner

Hanteringen av funktioner har blivit säkrare i C++ jämfört med C. I C antog exempelvis kompilatorn att en funktions returvärde var av typen int, om man hade glömt att ange returvärdestyp. I C++ kräver kompilatorn att det alltid *måste* finnas returvärdestyp angivet (void om inget returvärde).

Samma säkerhet är också inbyggd i parameteröverföringen. Alla formella parametrar *måste* ha en typdeklaration. Det går ej att som i C utelämna parameterdeklarationer varvid kompilatorn antar att en int överförs.

Ex : Skriv en funktion i C++ som tar en flyttals-vektor och antalet element i vektorn som parametrar och returnerar vektorns största element.

```
// maxberäkningsfunktion

float vekmax(float vektor[], int antal)
{
   float maxtal = vektor[0];

   for ( int i = 1; i < antal; i++ )
   {
      if ( vektor[i] > maxtal )
         maxtal = vektor[i];
   }

   return (maxtal);
}
```

```
// huvudprogram

void main()
{
   float max, vek[100];
   int nr;

   ......

   max = vekmax(vek, nr);

   ......
}
```

Utelämnar vi float framför funktionsnamnet vekmax ger en C++-kompilator ett felmeddelande medan en C-kompilator antar att det returnerade värdet är ett heltal, vilket naturligtvis blir fel eftersom bitarna i minnet (stacken) då tolkas som ett heltal men i verkligheten är ditsatta som koden för ett flyttal.

Hade vi utelämnat parametertypen float vektor[] (eller float *vektor, vilket är samma sak) så hade C++-kompilatorn gett ett felmeddelande medan en C-kompilator antagit att vi skickar en adress till ett heltal istället för en adress till ett flyttal. Detta ger fel vid pekaruppräkningen eller indexeringen, om int och float upptar olika antal byte i minnet.

En följd av att C++ inte förutsätter något om parametertyper är att en funktion *måste* vara , som ovan, definierad *innan* den kan anropas. Finns ej funktionens definition, i form av källkod längre upp i samma källkodsmodul, måste man ha en *funktionsdeklaration*, som innehåller funktionens prototyp(funktionshuvud), före anropet. I prototypen kan man utelämna parametrarnas namn och bara ange typen. För funktionen vekmax kan prototypen skrivas som :

```
float vekmax(float vek[], int antal);
```

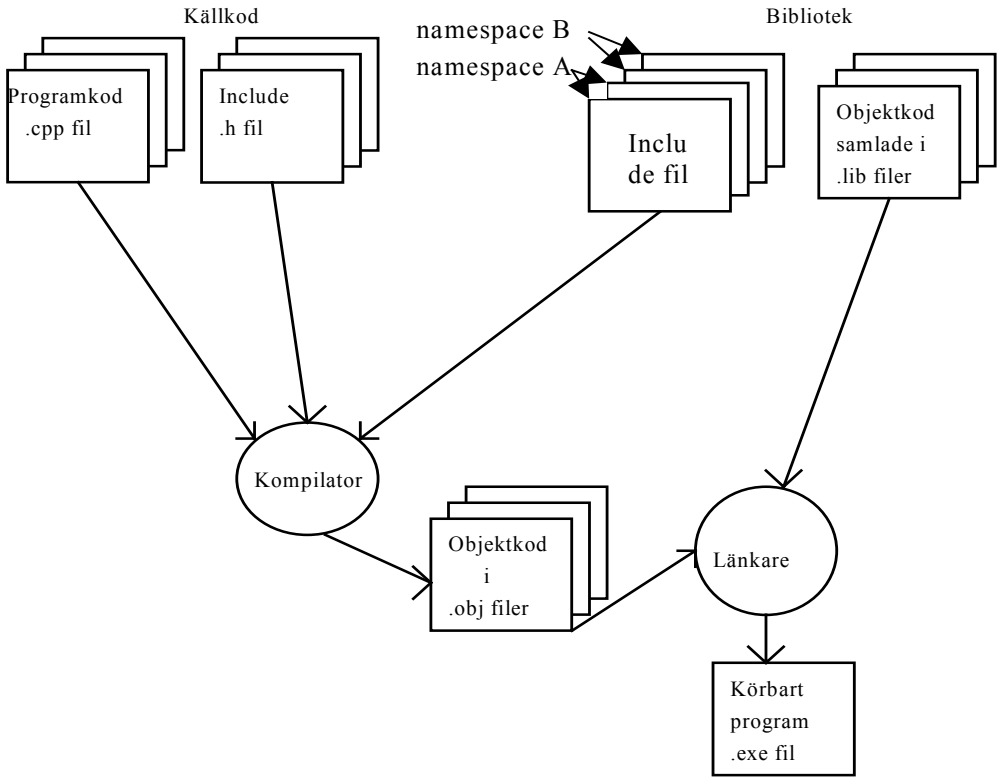
eller

```
float vekmax(float *, int);
```

OBS! Semikolon istället för funktionskropp.

Vid programmering, när man bestämt sig för hur man ska dela upp sitt program i moduler, skriver man in datatyper och tillhörande operationers funktionsprototyper i en headerfil modul.h, som sedan inkluderas till *både* modulens källkod och anropande program.

Standardbiblioteken är gjorda på detta sätt. I exempelvis string.h finns prototyper för strängfunktionerna strcpy etc. Själva koden hämtas, i form av objketkod, från ett bibliotek vid länkningen.



Vid *objektorienterad* programmering kommer modulerna att utgöras av klasser där både data och operationer finns. Klassens specifikation kommer då att finnas i en headerfil class.h och klassens implementation i en kodfil, class.cpp (se nedan).

2.3.1 Defaultvärden för parametrar

I C++ kan man ange *defaultvärden*, dvs värden som antas om ej annat anges, till de formella parametrarna.

Ex : Skriv en funktion som tar parametrar i form av en heltalsträng och heltalets bas samt returnerar heltalets värde. Defaultvärde på basen ska vara 10.

```
#include <cstring>

// Funktionen omvandlar str till ett tal i basen bas

int omvandla(char *str, int bas = 10)
{
   int tal = 0;

   for ( int i = 0; i < strlen(str); i++ )
      tal = tal * bas + str[i] - '0';

   return (tal);
}


// Huvudprogram

void main()
{
   int tal10, tal2;

   ......

   tal10 = omvandla("234");         // bas = 10

   tal2 = omvandla("1011", 2);      // bas = 2

   ......
}
```

Defaultvärden får endast ges till parametrar *i slutet* på parameterlistan. Detta innebär att man vid anropet enbart kan utelämna parametervärden på slutet och ej mitt i den aktuella parameterlistan.

2.3.2 Ellipsnotation

Med ellipsnotationen, ... (3 punkter) i den formella parameterlistan, kan man för en funktion ange att den tar ett variabelt antal parametrar. I stdarg.h finns ett antal funktioner och macron med vars hjälp man kan fiska upp de aktuella parametervärdena.

Ex : En popupmeny vars parametrar (argument) är bredd , höjd och ett okänt antal strängar, en sträng för varje menyalternativ, samt en tom sträng som avslutning.

```
#include <cstdarg>      //va_list, va_start, ..
#include <iostream>
using namespace std;

void popup(int b, int h, ...)
{
   va_list ap;
   char *cp;

   .....                        // popa upp menyn

   va_start(ap, h);             // ap pekar på första argumentet efter h
   cp = va_arg(ap, char *);     // fiska upp nästa sträng-argument
   while ( *cp != '\0' )        // så länge ej tom sträng
   {
      .....                     // skriv strängen cp in i menyn

      cp = va_arg(ap,char *); // fiska upp nästa sträng-argument
   }
   va_end(ap);                  // avsluta fiskandet
}



void main()
{
   ......

   popup(100, 400, "NEW", "OPEN", "SAVE", "");

   ......
}
```

När man använder ellipsnotation kopplar man bort typkontrollen vid parameteröverföringen. Det är nu programmerarens ansvar att se till att man tar emot argumenten på samma sätt som de skickas.

Detta är ett farligt sätt att programmera. Det finns ofta bättre sätt. I det här exemplet kunde alternativen funnits i en vektor av strängar och man hade skickat över vektorn och antalet element i vektorn.

Ex : Skriv ovanstående popupmeny utan att använda ellipsnotation.

```
#include <iostream>
using namespace std;

void popup(int b, int h, char *alt[], int nralt)
{
   .....                        // popa upp menyn

   for (int i = 0; i < nralt; i++)

      .....                     // skriv strängen alt[i] in i menyn

}



void main()
{
   char *menyalt[] ={ "NEW", "OPEN", "SAVE"};

   .....

   popup(100, 400, menyalt, 3);

   .....
}
```

2.3.3 Parametrar till main

Huvudfunktionen main är alltid den del av en modul som exekveringen startar i. Main kan ses som vilken C eller C++-funktion som helst, men med den skillnaden att den anropas av operativsystemet vid exekveringsstart. Man kan även ha parametrar till main. De aktuella parametrarna anges då tillsammans med *startkommandot* för programmet.

Ex : Skriv ett C-program, ccopy, som kopierar textfilen test.txt till filen ctest.txt. Filnamnen ska anges som parametrar vid programkörning enligt :

```
ccopy test.txt ctest.txt
```

```
// Huvudprogram -- ccopy.c

#include <stdio.h>

void main(int argc, char *argv[])
{
   FILE *infil, *utfil;
   char ch;

   // öppna filerna
   infil = fopen(argv[1], "r");
   utfil = fopen(argv[2], "w");

   // kopiera över alla tecken
   while ( (ch = fgetc(infil)) != EOF )
      fputc(ch, utfil);

   // stäng filerna
   fclose(infil);
   fclose(utfil);

}
```

Den första parametern, som formellt brukar betecknas argc, räknar det totala antalet strängar på kommandoraden. I ovanstående exempel blir argc = 3. Den andra parametern är en array av strängar. Strängarna i arrayen utgör de som finns på kommandoraden exv är argv[0] ="ccopy".

2.3.4 Referenser

I ANSI-C överförs parametrar enbart med *värdeanrop* (call-by-value). Detta innebär att vid ett anrop kopieras den aktuella parameterns *värde* till minnet (stacken). I den anropade funktionen kommer då den formella parametern att vara namnet på denna kopia. Vill man att funktionen ska ändra värdet på den aktuella parametern, måste man kopiera över en adress till denna. Den formella parametern, som då är en pekare (adressvariabel), kan då komma åt den aktuella parametern och eventuellt ändra dess värde.

Ex : Skriv en funktion i C, som beräknar minsta och största element i en heltalsvektor. Skriv också ett huvudprogram som anropar funktionen.

```
#include <stdio.h>

void vek_min_max_C(int vektor[], int antal, int *minp, int *maxp)
{
   int i;

   *minp = *maxp = vektor[0];

   for ( i = 1; i < antal; i++ )
   {
      if ( vektor[i] < *minp )
         *minp = vektor [i];
      else if ( vektor[i] > *maxp )
         *maxp = vektor[i];
   }
}


void main()
{
   int vek[] = {23, 34, 12, 56, 78};
   int min, max;

   vek_min_max_C(vek, 5, &min, &max);

   printf("Minsta talet : %d\n", min);
   printf("Största talet : %d\n", max);
}
```

OBS! Anropet med adresserna &min och &max som aktuella parametrar och motsvarande formella parametrar i form av pekarna minp resp maxp.

Bland annat för att på ett enklare sätt få tillbaka information från en funktion via parametrar, har man i C++ infört *referensvariabler*. Om en variabel deklareras som en referens till en annan variabel kommer båda att vara *namn på samma minnesutrymme*. Referensmarkeras en formell parameter kommer denna att vara ett namn på samma minnesutrymme som den aktuella parametern.

Ex : Skriv ovanstående min_max funktion i C++.

```
#include <iostream>
using namespace std;

void vek_min_max_CPP(int vektor[], int antal, int &minr, int &maxr)
{
   int i;

   minr = maxr = vektor[0];

   for ( i = 1; i < antal; i++ )
   {
      if ( vektor[i] < minr )
         minr = vektor [i];
      else if ( vektor[i] > maxr )
         maxr = vektor[i];
   }
}


void main()
{
   int vek[] = {23, 34, 12, 56, 78};
   int min, max;

   vek_min_max_CPP(vek, 5, min, max);

   cout << "Minsta talet : " << min << endl;
   cout << "Största talet : " << max << endl;

}
```

OBS! Anropet med variablerna min och max som aktuella parametrar och motsvarande formella parametrar i form av referenserna minr resp maxr. Referenser kan man se som alternativa namn (alias) på samma minnesutrymme. I ovanstående exempel kommer alltså min och minr att vara namn på samma variabel. Detsamma gäller max och maxr.

Även returvärden från funktioner kan vara referensdeklarerade. Detta medför att returvärdet blir en bestående variabel och att man exempelvis kan tilldela den värden.

Ex : Du vill arbeta med objektet bok och är intresserad av författare och antal sidor.

```
// specifikationsfil -- bok.h

struct bok
{
   char forfattare[40];
   int sidor;
};

bok &tjockast(bok &a, bok &b);
.....


// implementationsfil -- bok.cpp

#include "bok.h"

bok &tjockast(bok &a, bok &b)
{
   if ( a.sidor >= b.sidor)
      return a;
   else
      return b;
}

// huvudprogram -- bokmain.cpp

#include <iostream>
using namespace std;

#include "bok.h"

void main()
{
   bok minbok = {"Ulla", 125};
   bok dinbok = {"Per", 200};

   cout << tjockast(minbok, dinbok).forfattare << endl;
   tjockast(minbok, dinbok).sidor -= 1;
}
```

OBS! Funktionen tjockast returnerar ej bara ett värde utan ett *helt minnesutrymme*, som heter tjockast och som vi kan arbeta med som med vilken variabel som helst.

OBS! De formella parametrarna till funktionen tjockast måste också vara referenser i detta fall, ty annars hade vi försökt returnera ett *lokalt minnesutrymme* som endast existerar inuti funktionen. Är a och b istället referenser är det bara nya namn på de aktuella parametrarna minbok resp dinbok och tjockast kommer att vara en referens till någon av dessa.

2.3.5 Funktionsöverlagring

I C++ kan funktioner och operatorer överlagras, dvs ha samma namn men gälla för olika typer av parametrar (argument). Vilken av funktionerna som anropas bestäms av antalet och typerna på argumenten. Kompilatorn söker upp den funktion vars namn och formella typer i parameterlistan överensstämmer med de aktuella. Returparametern ingår ej i jämförelsen. Hittas ingen sådan funktion eller hittas flera sådana funktioner fås kompileringsfel.

Ex : Skriv två funktioner i C++ som returnerar en cirkels area. Den första funktionen ska ha radien som ett reellt tal som parameter medan den andra har radien i strängform som parameter.

```
#include <iostream>
using namespace std;

#include <cmath>


float cirkelarea(float radie)
{
   const float pi = 3.14159;


   return (pi * radie * radie);
}


float cirkelarea(char *radie)
{
   float r;
   const float pi = 3.14159;

   // konvertera sträng till reellt tal
   r = atof(radie);

   return (pi * r * r);
}


void main()
{
   float r;
   char sr[10];

   cout << "Ge radie : ";
   cin >> r;
   cout << "Arean = " << cirkelarea(r) << endl;

   cout << "Ge radie : ";
   cin >> sr;
   cout << "Arean = " << cirkelarea(sr) << endl;
}
```

Vid sökningen efter rätt överlagrad funktion sker också typkonverteringar enligt :

1) *Finn en funktion som överensstämmer exakt med argumentantal och argumenttyp*.
2) *Genomför typkonvertering enligt standardregler*.
3) *Genomför typkonvertering enligt egna regler (se nedan)*.

Hittar man ingen efter dessa sökningar fås kompileringsfel.

Operatorer som +, - etc kan betraktas som funktioner (se nedan) och kan alltså också överlagras. Man kan definiera + -operatorn för komplexa tal osv.

2.3.6 Inline-funktioner

När man skriver korta funktioner som anropas ofta kan man rationalisera bort anropet, som tar en viss tid och istället be kompilatorn att stoppa in funktionskoden på anropets plats, *inline*. Detta åstadkoms genom att sätta nyckelordet inline före funktionsnamnet.

Ex :

Skriv en inlinefunktion som omvandlar en liten bokstav till motsvarande stor och använd funktionen för att omvandla alla bokstäver till stora, för ett inläst namn.

```
#include <isotream>
#include <cstring>

using namespace std;

inline char storbokstav(char ch)
{
   if ( ch >= 'a' && ch <= 'z')
      return ch - 32;

   return ch;
}


void main()
{
   char namn[80];

   // läs max 80 tecken
   cout << "Ge namn : ";
   cin.getline(namn, 80);

   // gör om till stora
   for (int i = 0; i < strlen(namn); i++ )
      namn[i] = storbokstav(namn[i]);

   cout << namn;
}
```

OBS! Koden för storbokstav instoppas där den anropas