# 7 Templates och Exceptions

I senare versioner av C++ har man infört templates eller typparametrisering och exceptions eller undantagshantering.

Med hjälp av templates eller typparmetrisering kan man skriva *generella funktioner eller klasser* som man kan instansiera till att gälla för *olika datatyper*. På detta sätt sparar man kod och kan utnyttja samma kod för olika typer av data.

Med hjälp av exceptions kan man *flagga för oväntade fel* och *ta hand om dessa* på ett lämpligt sätt.

## 7.1 Templates

Template betyder schablon, mall eller mönster. Som programmerare skriver man *en mall för en funktion eller en klass*. Denna funktion eller klass kompileras inte utan utgör bara en mall. En användare kan sedan *instansiera funktionen eller klassen med önskad typ varvid kompilering sker.* På detta sätt kan man återanvända kod i flera tillämpningar. Man kan ha *en enda sorteringsfunktionsmall* som sorterar alla typer av data. Man kan ha *en enda stackklassmall* som kan stacka alla typer av data.

### 7.1.1 Funktionsmallar

Ex : Skriv en template för en urvals-sorteringsfunktion som tar den datatyp som ska sorteras som parameter samt anropar denna med en slumpad heltals- och flyttalsvektor.

```
// specifikation och implementering av rutinerna - sort.h

template <class T>
void ursort( T *vek, int nr)
{
   T min;
   int i, j, minind;

   for (i = 0; i < nr-1; i++ )
   {
      min = vek[i];
      minind = i;
      for (j = i+1; j < nr; j++)
      {
         if (vek[j] < min)
         {
            min = vek[j];
            minind = j;
         }
      }
      vek[minind] = vek[i];
      vek[i] = min;
   }
}
```

```
// Huvudprogram för sortering av heltalsvektor -- sortmai1.cpp


#include "sort.h"
#include <iostream>
#include <ctime>       // srand utnyttjar time
#include <cstdlib>     // srand, rand
using namespace std;


void main()
{
   int ivek[100];


   // slumpa vektorn

   randomize();
   for ( int i = 0; i < 100; i++ )
      ivek[i] = 100 + random(900);


   // sortera

   ursort(ivek, 100);


   // skriv ut vektorn med 10 tal per rad

   for ( i = 0; i < 100; i++ )
   {
      if ( i % 10 == 0)
         cout << endl;
      cout << ivek[i] << " ";
   }
}
```

OBS! Här behövs ingen *explicit instansiering* utan kompilatorn utgår från anropet av templatefunktionen ursort och kompilerar upp en heltalsversion av den. Man kan ej (i alla fall inte på ett enkelt sätt) använda separatkompilering för templates, utan man får se till att templatefunktionen finns i samma fil som anropet. Ovan har man templatefunktionens kropp i en headerfil som inkluderas så att den kompileras tillsammans med huvudprogrammet.

```
// Huvudprogram för sortering av flyttalsvektor-- sortmai2.cpp


#include "sort.h"
#include <iostream>
#include <ctime>
#include <cstdlib>
using namespace std;


void main()
{
   float fvek[100];


   // slumpa vektorn

   srand((unsigned)time(NULL)));
   for ( int i = 0; i < 100; i++ )
      fvek[i] = 100 + rand() % 2700 / 3.0;


   // sortera vektorn

   ursort(fvek, 100);


   // skriv ut vektorn med 10 tal per rad

   cout << endl << endl;
   cout.precision(2);
   cout.setf(ios::showpoint);
   for ( i = 0; i < 100; i++ )
   {
      if ( i % 10 == 0)
         cout << endl;
      cout << fvek[i] << " ";
   }
}
```

OBS! Mallen för sorteringsfunktionen ursort tas in med headerfilen och anropas med en flyttalsvektor. Inga speciella åtgärder krävs för att instansiera funktionen för flyttal utan detta klarar kompilatorn av på egen hand.

Alla datatyper eller klasser, som man ska sortera med mallen ursort ovan, måste ha de funktioner eller operatorer, som utnyttjas i funktionen, implementerade. Tittar man på ursort så ser man att data av typen T måste ha <-operatorn och =-operatorn implementerade.

Ex: RTAL-klassen har en default =-operator men saknar <-operator. Implementera <-operatorn samt använd mallen ursort för att sortera en RTAL-vektor.

```
// Specifikation av klassen RTAL -- rtal.h

#include <iostream>
using namespace std;

class RTAL
{
   private :
      int t;
      int n;
   public :
      RTAL(int t = 1, int n = 1);
      friend istream &operator>>(istream &in, RTAL &rt);
      friend ostream &operator<<(ostream &out, RTAL rt);
      RTAL forkorta();
      RTAL operator+(RTAL rt);
      RTAL operator-(RTAL rt);
      RTAL operator*(RTAL rt);
      RTAL operator/(RTAL rt);
      int operator<(RTAL rt);            // Mindre-än-funktion
};
```

```
// Implementation av klassen RTAL -- rtal.cpp

#include <iostream>
using namespace std;
#include "rtal.h"


.......


int RTAL::operator<(RTAL rt)
{
   return ( float(t)/n < douhble(rt.t)/rt.n );
}


........
```

```
// Huvudprogram -- rtalmain.cpp


#include "sort.h"
#include "rtal.h"


void main()
{


   // Skapa en vektor av rationella tal

   RTAL rvek[5] = {RTAL(1,2), RTAL(2,3), RTAL(1,3),
                   RTAL(5,6), RTAL(3,4)};


   // Sortera

   ursort(rvek, 5);


   // Skriv ut

   for ( int i = 0; i < 5; i++ )
      cout << rvek[i];
}
```

OBS! Man måste anropa RTAL-konstruktorn med angivna parametrar för varje element i vektorn.

OBS! Samma sorteringsrutin används som för heltal och flyttal tidigare. Kompilatorn kompilerar upp en korrekt version av ursort automatiskt.

### 7.1.2 Klassmallar

Man kan även typparametrisera sina klasser. På detta sätt fås betydligt mer generella klasser och man sparar mycket kodskrivande.

Ex : Skriv en mall för en klass som hanterar en stack i form av en vektor med aktuellt index som pekar på stackens topp. Klassen skall vara en mall med den datatyp som ska stackas och storleken på stacken som parametrar.

```
// Specifikation och implementation av klassen STACK -- stack.h

template <class T, int size>
class STACK
{
   private:
      int top;
      T vec[size];
   public:
      STACK();
      ~STACK();
      void push(T);
      T pop();
      int is_empty();
      int is_full();
};


template <class T, int size>
STACK<T, size>::STACK()
{
   top = -1;
}

template <class T, int size>
STACK<T, size>::~STACK()
{
}

template <class T, int size>
void STACK<T, size>::push(T data)
{
   vec[++top] = data;
}

template <class T, int size>
T STACK<T, size>::pop()
{
   return (vec[top--]);
}
template <class T, int size>
int STACK<T, size>::is_empty()
{
   return (top == -1);
}
template <class T, int size>
int STACK<T, size>::is_full()
{
   return (top == size - 1);
}
```

```
// Huvudprogram för att stacka tecken -- stackmain.cpp


#include <iostream>
using namespace std;
#include "stack.h"


void main()
{
   // Instansiera en stack för max 10 tecken

   STACK<char, 10> ch_stack;

   char ch;


   // läs och stacka tecken

   cout << " Skriv en text som avslutas med CTRL/Z" << endl;

   while ( !cin.get(ch).eof() )
   {
      if ( !ch_stack.is_full() )
      {
         ch_stack.push(ch);
      }
      else
      {
         cout << "Full stack!" << endl;
         break;
      }
   }


   // skriv ut stackade tecken

   while ( !ch_stack.is_empty() )
      cout << ch_stack.pop();
}
```

OBS! Man kan ha fler än en parameter till en template och parametrarna kan vara både vanliga datatyper, klasser eller värden.

```
// Huvudprogram för att stacka rationella tal -- stackmain.cpp


#include <iostream>
#include "stack.h"
#include "rtal.h"
using namespace std;


void main()
{
   // Instansiera en stack för max 5 rationella tal

   STACK<RTAL, 5> r_stack;

   RTAL r;


   // läs och stacka rationella tal

   cout << "Avsluta med ett negativt bråk !" << endl;
   cin >> r;
   while ( !(r < 0) )
   {
      if ( !r_stack.is_full() )
      {
         r_stack.push(r);
      }
      else
      {
         cout << "Full stack!" << endl;
         break;
      }
      cin >> r;
   }


   // skriv ut stackade rationella tal

   while ( !r_stack.is_empty() )
      cout << r_stack.pop();
}
```

OBS! Inläsningen avslutas med ett negativt bråk. Eftersom vi inte har >-operatorn implementerad så måste vi använda <-operatorn. I anropet r < 0 omvandlas först 0 till det rationella talet 0/1, med en av RTAL-klassens konstruktorer.

## 7.2 Klassbibliotek

Klassbibliotek finns i de flesta C++-miljöer för att exempelvis hantera samlingar av objekt (containerklasser), användargränssnitt med fönsterhantering, matematiska funktioner, realtidsprogrammering m.m. Dessa klassbibliotek bygger mycket på arvshierarkier. När det gäller exempelvis containerklasserna har man dock i senare versioner mer och mer övergått till att använda typparametrisering.

### 7.2.1 Containerklasser

*Containrar*, samlingar eller behållare, är klasser för att samla och organisera andra objekt i form av listor, vektorer, binära träd , hashtabeller etc. För att hålla reda på vilket objekt man för närvarande jobbar med i behållaren använder man speciella *iteratorobjekt* (indexering eller pekare). Om iteratorn skulle ingå som en del av containern skulle man bara kunna peka på ett aktuellt objekt åt gången. Är däremot iteratorn ett eget objekt kan man definiera flera iteratorer och därmed hålla reda på flera aktuella objekt åt gången.

Antag att containern representerar tavlorna på ett konstmuseum och att iteratorn håller reda på vilken tavla som en besökare just nu tittar på. Om iteratorn skulle ingå i containerklassen skulle museet bara kunna tillåta en besökare åt gången. Har man däremot en separat iteratorklass kan man definiera ett iteratorobjekt för varje besökare.

En container för en länkad lista av tavlor och en länkad lista av VVS-objekt ska tillhandahålla samma operationer, som att addera nya objekt, ta bort objekt osv. precis som motsvarande iteratorer ska ha operationer, som peka på första, peka på nästa osv. Här är det alltså mycket lämpligt att skriva klasserna som templates och sedan instansiera dessa för olika typer eller klasser.

I den nya C++-standarden ingår ett klassbibliotek för containerklasser som består av klass- och funktionsmallar. Biblioteksdesignen grundar sig på begreppen *container, iterator och algoritm.* Dessa kan sedan kombineras på olika sätt och de representerar på sätt och vis tre oberoende komponenter vid konstruktionen.

Standardbiblioteket innehåller containrar som exempelvis vector, list, stack och dqueue, iteratorer som indexering, next, forward och random och algoritmer som find, reverse, rotate och sort.

Ex : Skriv ett program som utnyttjar det nya standardbibliotekt i C++ för att läsa in ett antal heltal från filen tal.txt till en vektor som sorteras och skrivs ut..

```
#include <vector>
#include <fstream>

using namespace std;    // För att komma åt biblioteket direkt

void main()
{

   // Instansiera en dynamisk vektor-container för heltal
   vector <int> v;


   ifstream f("tal.txt");
   int tal;


   while (!(f >> tal).eof())
   {
      // Anropa container-funktion och iterator-funktion
      // Stoppar in talen sist i vektorn
      v.insert(v.end(), tal);
   }


   // Anropa algoritm-funktion och iteratorfunktioner

   // Sortera
   sort(v.begin(), v.end());


   // Skriv ut
   for (int i = 0; i < v.size(); i++)
      cout << v[i] << endl;

}
```

OBS! Vektorn dimensioneras automatiskt av biblioteket. En användare behöver inte bry sig om varken allokering eller avallokering.

Ex : I kylsystemet kan man istället för att själv göra en lista av VVS-komponenter använda sig av det standardiserade C++-biblioteket för att skapa listan av komponenter. Skissa på hur detta ska göras.

```
#include <list>

using namespace std;

list <VVS *> komplista;// En lista av pekare till VVS-komponenter


// Sätt in komponenternas adresser i listan
komplista.insert(komplista.begin(), &n1);
komplista.insert(komplista.begin(), &n2);
komplista.insert(komplista.begin(), &n3);
komplista.insert(komplista.begin(), &v1);
komplista.insert(komplista.begin(), &v2);



// Händelsefunktionen, som hanterar omritning av fönstret, ska nu
// kunna utnyttja listan, för att rita alla komponenter.


list <VVS *>::iterator iter;  // Definition av en iterator


// Rita alla komponenter

for (iter = komplista.begin(); iter != komplista.end(); iter++)
   {
      (*iter)->rita();
   }
}
```

OBS! Iteratorn iter är en pekare till en VVS-pekare. Avrefererar man iter så får man en pekare som man måste avreferera en gång till för att få komponenten.

OBS! Fördelen med att utnyttja klassbiblioteket, för att skapa en lista, är att man slipper att gå in i de olika komponentklasserna och göra förändringar.

## 7.3 Exceptions

Exceptions eller undantag är ett sätt som införts i senare versioner av C++ för att ta hand om *oväntade fel* som slut på minne, fel på printer etc. Ett väntat fel som en felaktig inmatning ska man inte betrakta som ett undantag utan sådana fel ska inmatningsrutinens filter ta hand om.

Undantagshantering är en metod att hantera fel på annan plats än där det upptäcks. När felet uppstår kastas ett undantag (skjuts en nödraket ) och sedan är det upp till användaren eller klienten att tillverka en undantagshanterare som tar hand om felet på lämpligt sätt.

Har printern lagt av ska printerobjektet resa ett undantag och som användare kan man då få tillbaka kontrollen och exempelvis skriva ut på en annan printer.

I vanlig ANSI-C kan man skicka nödraketer men det är svårare att få tillbaka kontrollen och göra något vettigt åt felet. Den metod som man brukar använda är att utnyttja funktionsvärdet som en test på om allting har fungerat då funktionen körs. Detta går bra så länge inte funktionsvärdet är upptaget av ett verkligt retur-värde.

Ex : Om vi betraktar mindre-än-funktionen för RTAL-objekt så kan det uppstå ett fel i form av att nämnaren är noll vid divisionen.

```
.....

int RTAL::operator<(RTAL rt)
{
   return ( float(t)/n < float(rt.t)/rt.n );
}

.....

template <class T>
void ursort( T *vek, int nr)
{
   T min;
   int i, j, minind;

   for (i = 0; i < nr-1; i++ )
   {
      min = vek[i];
      minind = i;
      for (j = i+1; j < nr; j++)
      {
         if (vek[j] < min)
         {
            min = vek[j];
            minind = j;
         }
      }
      vek[minind] = vek[i];
      vek[i] = min;
   }
}
```

```
// Huvudprogram -- rtalmain.cpp

#include "sort.h"
#include "rtal.h"

void main()
{

   // Skapa en vektor av rationella tal

   RTAL rvek[5] = {RTAL(1,2), RTAL(2,3), RTAL(1,3),
                   RTAL(5,0), RTAL(3,4)};      //OBS! 0 i nämnaren


   // Sortera

   ursort(rvek, 5);


   // Skriv ut

   for ( int i = 0; i < 5; i++ )
      cout << rvek[i];
}
```

Kör man programmet kommer man att åka ur med ett meddelande enligt :

Floating point error : Divide by zero
Abnormal programtermination

Felet uppstår i <-funktionen. Vill man ha bättre kontroll på läget kan man sätta in en test enligt :

```
int RTAL::operator<(RTAL rt)
{

   if (n*rt.n == 0)
   {
      cerr << "Division med 0 i <-funktionen" << endl;
      exit(1);
   }
   return ( float(t)/n < float(rt.t)/rt.n );
}
```

Kör man programmet nu åker man ur men får en bättre feldiagnos i form av texten :
Division med 0 i <-funktionen

Istället för exit kan man använda funktionen assert, som finns i assert.h, och kräva en *försäkran* av användaren om att man inte skickar nämnare som är 0 till <-funktionen enligt :

```
int RTAL::operator<(RTAL rt)
{
   assert(n*rt.n > 0 );
   return ( float(t)/n < float(rt.t)/rt.n );
}
```

Nu terminerar programmet med :

Assertion failed : n*rt.n > 0, file rtal.cpp line 111
Abnormal programtermination

Ovanstående metoder skickar nödraketer men som användare kan man inte göra något, *utan programmet terminerar*. Det är här som den nya exception-modellen kommer in. I <-funktionen kastar man ett undantag om någon av nämnarna är 0 enligt :

```
int RTAL::operator<(RTAL rt) throw(const char *) // OBS!
{

   if (n*rt.n == 0)
   {
      // Kasta undantag
      throw("Division med 0 i <-funktionen");

   }
   return ( float(t)/n < float(rt.t)/rt.n );
}
```

Det är sedan upp till användaren av klassen att fånga upp undantaget. Vilka undantag som kan kastas framgår av att man *efter funktionshuvudet* skriver ut dessa med angiven parametertyp. När det gäller medlemsfunktioner i en klass ska man skriva in vilka undantag resp medlemsfunktion kastar i klasspecifikationen. Den som använder <-funktionen ska alltså fånga in ett felmeddelande i form av en sträng. Den anropande funktionen är i detta fall ursort. Här kan man ta hand om felet eller låta bli. Låter man bli transporteras felet vidare till nästa nivå osv.

Istället för att enbart kasta ett uttryck som exception kan man kasta ett antal standardfel som finns i klassen stdexcept eller dess subklasser. För att komma åt dessa standardfelklasser inkluderar man stdexcept.h. I ovanstående exempel hade man kastat overflow_error enligt:

```
      throw  overflow_error("Division med 0 i <-funktionen");
```

Antag att man tar hand om felet i ursort, skriver ut felmeddelandet och reser ett nytt undantag där man skriver in värdet på det index som felet kan ha uppstått i enligt :

```
template <class T>
void ursort( T *vek, int nr) throw(int)
{
   T min;
   int i, j, minind;

   for (i = 0; i < nr-1; i++ )
   {
      min = vek[i];
      minind = i;
      for (j = i+1; j < nr; j++)
      {
         // försök att anropa <-funktionen
         try
         {
            if (vek[j] < min)
            {
               min = vek[j];
               minind = j;
            }
         }

         // fånga ev undantag från <-funktionen
         catch(const char *mess)
         {
            cerr << mess << endl;

            // res nytt undantag
            throw(j);
         }
      }
      vek[minind] = vek[i];
      vek[i] = min;
   }
}
```

OBS! Hur man sätter in anropet av funktionen i ett try-block. Om inga undantag kastas från <-funktionen körs try-blocket rakt upp och ner. Om det däremot kastas ett undantag hamnar man i catch-blocket för vidare åtgärder. I ovanstående fall skriver man ut felmeddelandet och kastar ett nytt meddelande kompletterat med det index som orsakat felet.

Sedan gäller det att fånga undantaget i huvudprogrammet och hantera felet på lämpligt sätt, exempelvis genom att läsa in ett nytt värde till vektorn och sortera på nytt :

```
// Huvudprogram -- rtalmain.cpp

#include "sort.h"
#include "rtal.h"

void main()
{

   // Skapa en vektor av rationella tal

   RTAL rvek[5] = {RTAL(1,2), RTAL(2,3), RTAL(1,3),
                   RTAL(5,0), RTAL(3,4)}; //OBS! 0 i nämnaren


   // Sortera

   int done = 0;

   while ( !done )
   {
     try
      {
         ursort(rvek, 5);
         done = 1;
      }
      catch(int ind)
      {
         if ( ind == 1) // Kan vara fel i index 0
         {
            cout << "Gammmalt rvek[0] : " << rvek[0];
            cout << endl << "Ge nytt värde : ";
            cin >> rvek[0];
         }
         cout << "Gammalt rvek[" << ind << "] : " << rvek[ind];
         cout << endl << "Ge nytt värde : ";
         cin >> rvek[ind];
      }
   }

   // Skriv ut

   for ( int i = 0; i < 5; i++ )
      cout << rvek[i];
}
```

OBS! Finns inget passande catch-block propagerar undantaget uppåt för att slutligen hamna i operativsystemet. Ett catch-block med parameterlistan (...) fångar upp samtliga undantag.