1 Objektorienterad programkonstruktion

Ett program *bearbetar data* eller *information* inom något område av vår verklighet. Själva arbetet att skriva programmet, *implementationsfasen*, föregås alltid av först en *analysfas* och sedan en *designfas*. Det är i analys- och designfasen som själva programmet eller systemet *konstrueras*. Under implementationsfasen kodar man bara den färdiga konstruktionen. Efter implementationen fortsätter systemarbetet med *testfasen* och *underhållsfasen*.

1.1 Systemkonstruktion

Analysfasen ska ge svar på frågan *vad.* Vad ska ingå i systemet? Vad ska systemet göra? Till grund för analysfasen ligger en kravspecifikation från beställaren. Man börjar med att avgränsa den del av verkligheten som ska ingå i systemet samt bestämmer vilket informationsflöde som ska in och vilket som ska ut från detta avgränsade system. Man delar in systemet i delsystem och undersöker informationsflödet mellan delsystemen. Man har hela tiden kontakt med beställaren för att diskutera systemet och införa förändringar. Eventuellt tillverkar man en prototyp, exempelvis i form av ett användargränssnitt, för att ha som hjälp vid diskussionerna.
Slutdokumentet från analysen ska utgöra en detaljerad beskrivning av den verklighet som ska ingå i systemet.

Designfasen ska ge svar på frågan *hur*. Hur ska systemet förverkligas? Hur ska modellen, avbildningen, av verkligheten se ut? Vilken hårdvara ska användas? Vilket språk ska användas för mjukvaran? Hur ska man dela upp i moduler? Här förbereder man för implementationen. Man dukar bordet för att lättare kunna koda programmet. Här delar man också upp arbetet i projektgrupper där varje grupp exempelvis får ett delsystem att koda. Designfasen dokumenteras med en beskrivning av den modell som ska byggas samt en detaljerad plan på hur modellen ska förverkligas.

I *implementationsfasen* kodas programmet som konstruerats i designfasen. Det som kan återstå av konstruktionsarbete i denna fas är formuleringar av algoritmer och vissa detaljer i programkonstruktionen.

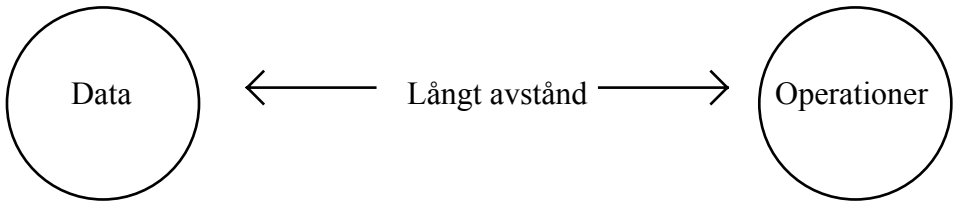
Slutligen återstår en *testfas*, där programmet testas först i delar inom varje grupp och sedan, efter hopsättning, hela systemet. När systemet är levererat följer *underhållsfasen* där fel ska åtgärdas och förändringar införas.

1.1.1 Traditionell konstruktion

Det *traditionella* sättet att angripa programkonstruktionsarbetet har varit att fråga sig, *vad ska systemet göra*. Man har inriktat sig på funktionaliteten, operationerna i systemet. Detta sätt att angripa arbetet har ofta misslyckats. Man har fått system som blivit *svåra att underhålla och att ändra i*. Den största delen av systemkostnaderna idag, utgör underhållskostnader.

Felet med det traditionella sättet är att man skapar stora avstånd mellan *data* och *bearbetningen* av denna data. Ändrar man på någon datatyp blir det svårt att hitta alla ställen i systemet som påverkas. Vill man ha in en ny funktion är det svårt att se hur denna funktion kommer att påverka resten av systemet.

Traditionella sättet :



Dessutom är det svårt att jämföra verkligheten med modellen eftersom verkligheten består av saker (objekt) och modellen av operationer. Det är svårt att föra diskussioner mellan de olika intressegrupperna under de olika systemeringsfaserna. Programmerarna pratar om algoritmer för bearbetningen och de tekniska experterna pratar om de tekniska objekten som finns i systemet Detta försvårar analys- och designarbetet.

Underhållsarbetet blir också betydligt svårare eftersom modellen i form av operationer är en dålig bild av verkligheten. Det är svårt att hitta rätt i modellen. Var ska ändringen göras? *Modell och verklighet passar dåligt ihop*.

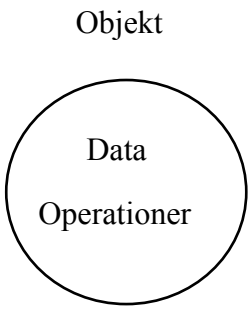
1.1.2 Objektorienterad konstruktion

Det nya sättet, *objektorientering,* angriper problemet genom att ställa frågan, *vad finns i systemet, vad ska systemet hantera*. Man inriktar sig på vad systemet ska bestå av istället för vad det ska göra. Det är oftast så att vad systemet består av är mera beständigt än vad systemet ska göra.

Vad består då systemen av? Svaret är *objekt*. Objekt kan vara bibliotekarie och böcker i ett bibliotekssystem, givare och instrument i ett mätsystem etc. Objekten har *tillstånd* i form av värden på data och dess tillstånd kan ändras med de till objekten hörande operationerna. En bok är utlånad och när den lämnas in övergår tillståndet utlånad till inne.

Fördelen med det objektorienterade synsättet är att man har bättre kontakt mellan data och operationer och därmed blir det lättare att underhålla och att ändra i systemen.

Objektorienterade sättet :

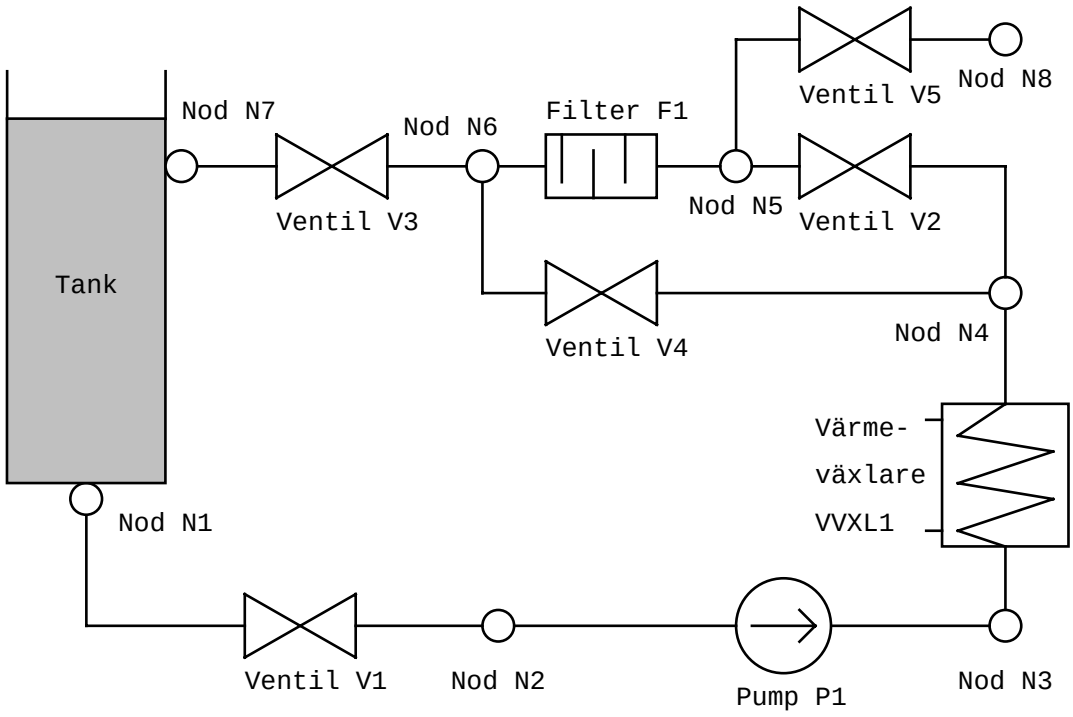


En annan stor fördel med objektorientering är att man kan prata om objekt och relationer mellan objekt redan i analysfasen och hela vägen ner till implementationsfasen och underhållsfasen. *Alla inblandade i systemkonstruktionen pratar samma språk, objektspråket*.

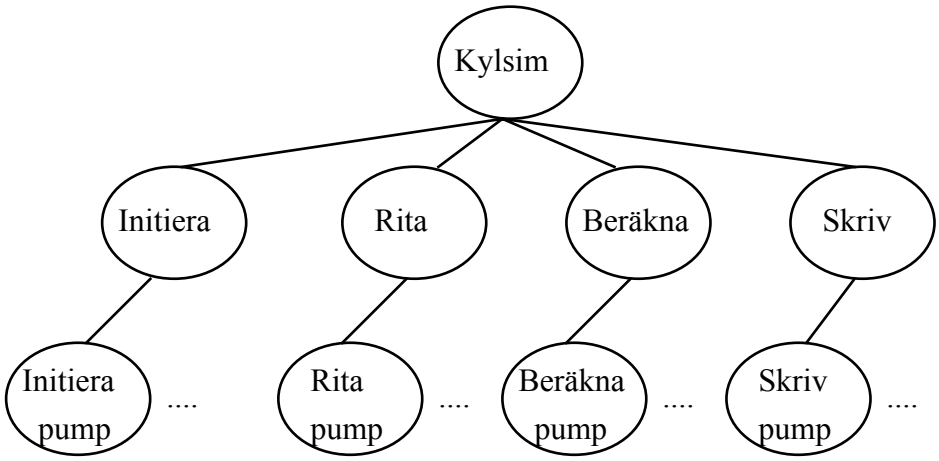
Den modell som man bygger upp liknar verkligheten betydligt mer än modellen i det traditionella sättet. Man har modeller för objekt som också existerar i verkligheten. I underhållsfasen är det lättare att hitta var ändringarna ska stoppas in eftersom *modellen liknar verkligheten*

1.1.3 Konstruktionsexempel, kylsimulering

Ex: Gör ett program som ska simulera ett kyl- och reningssystem för vattnet i en reaktortank i ett kärnkraftverk. Hett vatten cirkuleras med hjälp av en pump från tanken genom en värmeväxlare och ett filter tillbaka till tanken. Filtret blir med tiden igensatt varvid larm ska ges så att operatören kan rena filtret genom att backspola det. Systemet ska bestå av ett grafiskt användargränssnitt, MMI = Man Machine Interface, med vars hjälp man kan styra och övervaka systemet



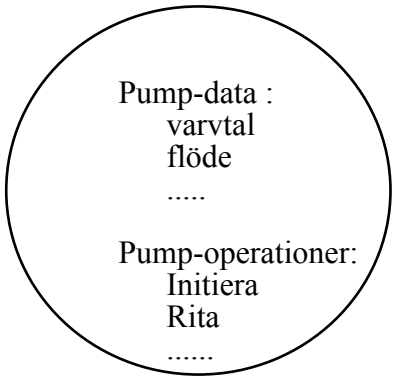
Det traditionella sättet att angripa problemet skulle vara att fråga sig vad systemet ska göra. Det ska initiera, rita, beräkna, skriva osv. Genom stegvis förfining kan man komma fram till en uppdelning i operationer enligt exempelvis:



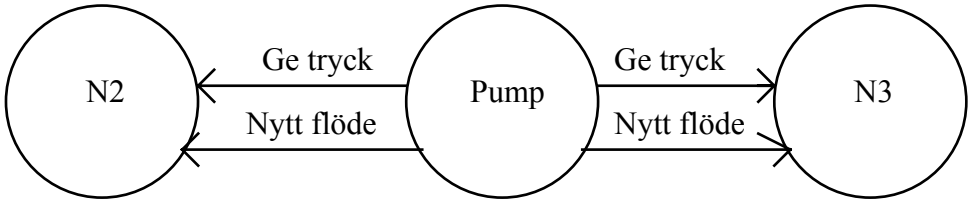
Här ser man nackdelen med det traditionella sättet. Datatypen för pumpen definieras i huvudprogrammet. Bearbetningen av pumpens data kommer däremot att spridas ut över ett stort antal filer och det blir svårt att leta upp alla ställen vid underhåll och förändringar. Detsamma gäller bearbetningarna av de andra komponenterna.
I verkligheten har man en pump som finns på ett ställe i systemet och då borde även modellen ha en pump där både data och operationer är samlade.

Med det nya sättet, objektorientering, frågar man sig vad ska systemet hantera? Vilka objekt finns i systemet? Man bygger sitt system kring dessa objekt och samlar objektens data och operationer på ett ställe.

I exemplet finns ett antal tydliga fysiska objekt som pump, filter etc. Varje objekts tillstånd bestäms av dess data (attribut) och funktionaliteten av dess operationer (metoder). Pumpens tillstånd bestäms av varvtal, flöde, pumpkonstanter mm och operationerna är initiera, rita, beräkna flöde, starta, stoppa etc.



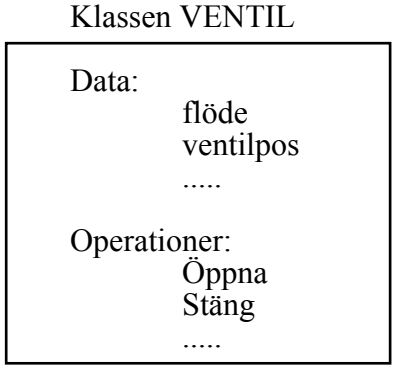
Det är dessa objekt som man använder som byggklossar när man bygger sitt system. *Funktionaliteten i systemet byggs upp av att objekten anropar eller skickar meddelanden till varandra.*

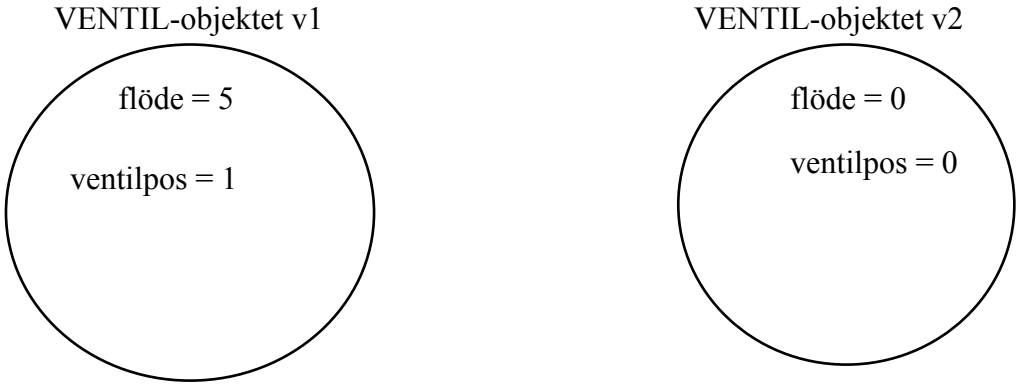


Pumpen frågar noderna N2 resp N3 om deras tryck. När den fått tryckskillnaden räknar den ut vad flödet ska bli utgående från sitt aktuella tillstånd vad gäller pumpkonstanter och varvtal. Det uträknade flödet skickas till noderna som uppdaterar sina tryck så att in- och utflödet till varje nod blir lika stort osv.

1.1.4 Klasser och objekt

Ventilerna och noderna, som är många, visar behovet av objekttyper, så kallade *klasser*. Man börjar med att skapa klassen ventil som beskriver vilken typ av data och operationer som ventiler ska ha. Objekten skapas sedan som *instanser* av denna klass med de aktuella värden på data som gäller just för denna ventil. *Metoderna (operationerna) är lika för alla objekt av samma klass medan data är individuella för varje objekt*. Objekt kan jämföras med vanliga variabler och klasser med typer.





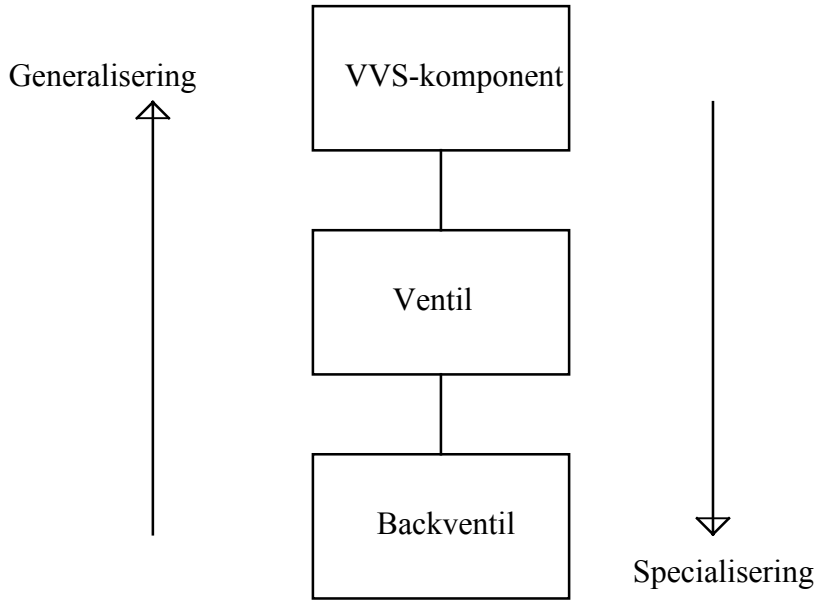
Att *hitta de intressanta objekten* i sitt system är en viktig del av systemarbetet. Det finns objekt i ett system som inte direkt är så påtagligt fysiska som ventil etc. Dessa objekt som popupmeny, länkade listor etc plockas in i systemet efter hand vid designen och implementationen, när behov uppstår.

1.1.5 Relationer

Vid systemarbetet letar man inte bara efter objekten utan minst lika viktigt är att hitta *relationer* mellan olika objekt eller mellan olika klasser.

Man letar exempelvis efter *arvsrelationer* i form av antingen *specialiseringar* eller *generaliseringar*. Hade man i kylsystemet haft en backventil skulle den vara en speciell ventil med egenskapen att flödet vore noll i backriktningen. Man kan även generalisera genom att samla alla komponenter ventil, pump etc till en samlingsklass VVS-komponent.

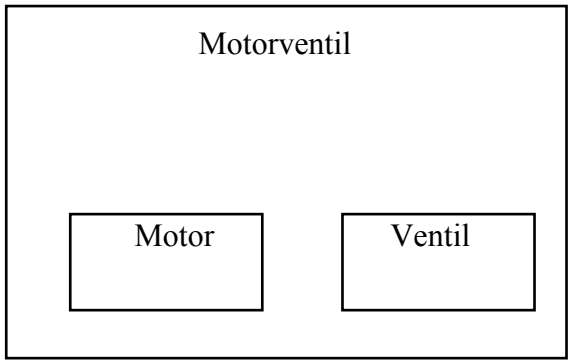
Arvsrelation :



En arvsrelation har man om man kan säga att ventil *är en* VVS-komponent, backventil *är en* ventil etc.

En annan vanlig relation mellan objekt är *aggregatrelationen*. Här letar man efter *helheten* eller *delarna*. En motorventil *består av* en motor och en ventil eller omvänt en motor och en ventil kan *sättas ihop* till en motorventil.

Aggregatrelation:



En aggregatrelation finns om man kan säga att en motorventil *består av* en motor, en motorventil *består* av en ventil osv.

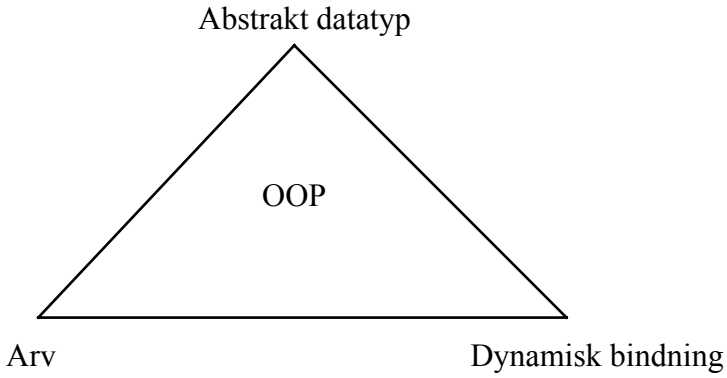
Slutligen letar man efter *associationsrelationer*. Här ska man titta på omgivningen till objekten och titta på hur objekt påverkar eller påverkas av varandra. En ventils flöde påverkas av sina noders tryck. Ett ventilobjekt måste alltså *känna till* sin in-nod och sin ut-nod för att kunna beräkna tryckskillnaden.

Associationsrelation:

Man har en associationsrelation om ventil *känner* till nod eller *refererar* till nod.

1.2 Objektorienterade programmeringsspråk

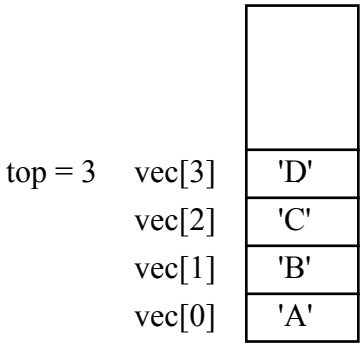
För att ett programmeringsspråk ska vara objektorienterat måste man i språket ha möjligheter att på ett enkelt sätt hantera de tre hörnstenarna i objektorientering nämligen :



1.2.1 Abstrakt datatyp

Abstrakta datatyper är det som man ovan har kallat för klasser. I en abstrakt datatyp *inkapslar* man data och operationer tillsammans. Målsättningen är att man ska ha nära mellan data och operationer. Man kan ha en bra inkapsling av data och operationer även i språk som Pascal och C, men i de objektorienterade språken finns specialkonstruktioner, som *klassbegreppet*, för detta. Klassen ger en *säkrare inkapsling* med ett inbyggt skydd mot manipulering av data.

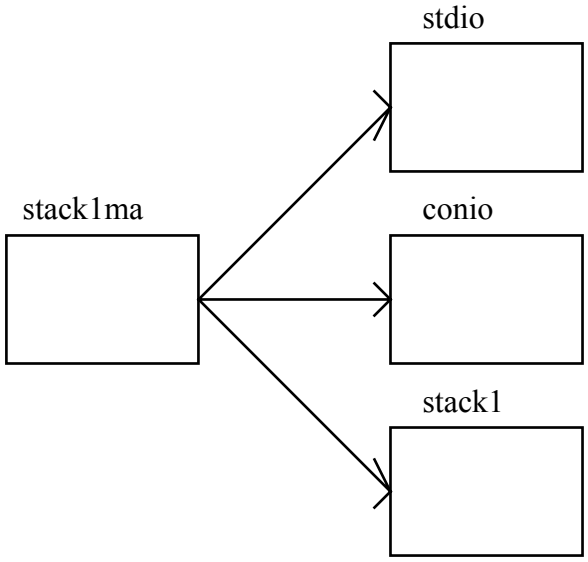
Ex : En stack är en lista av LIFO-typ dvs det sista elementet som stoppas in i listan tas ut först. Avbilda en stack av tecken som en abstrakt datatyp med data i form av en vektor av tecken och aktuellt index samt med operationer för att rensa, stoppa in, ta ut och kontrollera om tom stack eller full stack.


top = 3
vec[3]
'D'
vec[2]
'C'
vec[1]
'B'
vec[0]
'A'

a) I ANSI_C samlar man alla typdefinitioner och funktionsprototyper i en headerfil. Funktionskropparna skriver man i en implementationsfil där headerfilen inkluderas och som kompileras separat.

```
/* specifikation av stack -- stack1.h */

typedef struct
{
   int top;
   char vec[100];
} stack;

void clear(stack *sp);        /* Rensar stacken            */
void push(stack *sp, char c); /* Stoppar in c på stacken   */
char pop(stack *sp);          /* Tar tecken från stacken   */
int is_empty(stack s);        /* Är stacken tom?           */
int is_full(stack s);         /* Är stacken full?          */


/* implementation av stack -- stack1.c */

#include "stack1.h"

void clear(stack *sp)
{
   sp->top = -1;
}

void push(stack *sp, char c)
{
   sp->vec[++sp->top] = c;
}

char pop(stack *sp)
{
   return (sp->vec[sp->top--]);
}

int is_empty(stack s)
{
   return (s.top == -1);
}

int is_full(stack s)
{
   return (s.top == 99);
}
```

I huvudprogrammet inkluderar man headerfilen och kompilerar. Slutligen länkas huvudprogrammet ihop med funktionernas implementationsfil och ev biblioteksfiler och körs. Beroendediagrammet blir enligt :




```
/* huvudprogram -- stack1ma.c */

#include <stdio.h>
#include <conio.h>
#include "stack1.h"

void main()
{
   char ch;
   stack ch_stack;

   clear(&ch_stack);

   /* Läs in tecken och stacka */
   printf("\nSkriv en text som avslutas med CTRL/Z\n");
   while ((ch = getchar()) != EOF)
   {
      if ( !is_full(ch_stack) )
         push(&ch_stack, ch);
      else
      {
         printf("Full stack!\n");
         break;
      }
   }
   /* Töm och skriv ut stackens tecken */
   while ( !is_empty(ch_stack) )
      putchar(pop(&ch_stack));

   getch();
}
```

b) I C++ har man konstruktionen *klass* som direkt motsvarar den abstrakta datatypen. För klassen har man både en specifikationsdel med data och funktionsdeklarationer (huvuden) och en implementationsdel med funktionsdefinitioner (kroppar). Implementationsdelen kompileras separat liksom huvudprogrammet och länkas sedan ihop till en exekverbar fil.

```
// specifikation av klassen STACK -- stack2.h

class STACK
{
    private:
       int top;
       char vec[100];
    public:
       void clear();
       void push(char c);
       char pop();
       int is_empty();
       int is_full();
 };


// implementation av klassen STACK -- stack2.cpp

#include "stack2.h"

void STACK::clear()
{
   top = -1;
}

void STACK::push(char c)
{
   vec[++top] = c;
}

char STACK::pop()
{
   return (vec[top--]);
}

int STACK::is_empty()
{
   return (top == -1);
}

int STACK::is_full()
{
   return (top == 99);
}
```

```
// huvudprogram -- stack2ma.cpp

#include <conio.h>
#include "stack2.h"

#include <iostream>
using namespace std;

void main()
{
   char ch;
   STACK ch_stack;

   ch_stack.clear();

   // läser tecken och stoppar in dessa på stacken
   cout << " Skriv en text som avslutas med CTRL/Z\n";
   while ( (ch = cin.get()) != EOF )
   {
      if ( !ch_stack.is_full() )
         ch_stack.push(ch);
      else
      {
         cout << "Full stack!" << endl;
         break;
      }
   }

   // töm stacken och skriv ut alla tecken
   while ( !ch_stack.is_empty() )
      cout << ch_stack.pop();

   getch();
}
```

OBS! I ANSI-C definierar man en *variabel* ch_stack av *typen* stack och sedan anropar man stackoperationerna med denna variabel *som parameter*. Ska man ändra på stackvariabelns värde måste man skicka en *adress* som parameter, som i anropet push(&ch_stack, ch).

I C++ definierar man ett *objekt* ch_stack av *klassen* STACK och anropar detta objekts operationer med *punktnotation*. Man skickar meddelanden till objektet att göra olika saker. Exempelvis kommer meddelandet ch_stack.push(ch) att ändra tillståndet hos vårt objekt ch_stack, så att top ökas med ett och vec[top] tilldelas ch. Objektet ch_stack håller sedan reda på sitt tillstånd. Man kan fråga ch_stack om den är tom och den svarar nej.

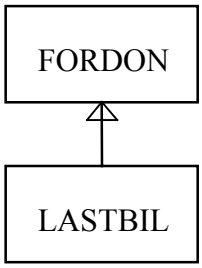
Olika objekt av samma klass kommer att innehålla olika data men använda klassens gemensamma operationer. Jfr vanliga variabler som float, int etc.

I C++ har man infört namespaces för att inte få namnkollisioner mellan egenupphittade namn och namn i de olika C++-biblioteken. Med using namespace kommer man åt namnen från Biblioteken utan att behöva kvalificera namnen med ex std::cout. Nytt i C++ är också att man inte ska använda .h för bibliotekens hederfiler utan bara för sina egna h-filer. Använder man gamla ANSI-C filer sätter man ett c framför som ex. #include <cstdio>.

1.2.2 Arv

Arv innebär att nya klasser kan specialiseras genom att man lägger till nya data och operationer utöver de redan befintliga i tidigare klasser. Man kan komponera en egen klass speciellt efter sina behov men man kan använda allt som gjorts tidigare. Objekt som ärver står i ett "is a" eller "är en" förhållande till det objekt från vilket man ärver. Egentligen sker arvet mellan klasser och då pratar man om *basklass* och *subklass*.

Ex: Ett fordon karakteriseras av data i form av registreringsnummer och ägare och operationer för att läsa in data och skriva ut data. En lastbil *är ett* fordon med tillägg av datadelen lastkapacitet. Skriv ett C++-program som läser in data till en lastbil och skriver ut motsvarande data på skärmen.




```
// specifikation av klassen FORDON -- fordon.h

class FORDON
{
   private:
      char regnr[10];
      char owner[40];
   public:
      void las();
      void skriv();
};


// implementation av klassen FORDON -- fordon.cpp

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

void FORDON::las()
{
   cout << endl << "Regnr : ";
   cin.getline(regnr, 10);
   cout << "Ägare : ";
   cin.getline(owner, 40);
}

void FORDON::skriv()
{
   cout << endl << regnr << endl;
   cout << owner << endl;
}
```

```
// specifikation av klassen LASTBIL -- lastbil.h

#include "fordon.h"

class LASTBIL:public FORDON        // OBS! Ärver FORDON
{
   private:
      float last;
   public:
      void las();
      void skriv();
};


// implementation av klassen LASTBIL --lastbil.cpp

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

void LASTBIL::las()
{
   FORDON::las();
   cout << "Lastvikt : ";
   cin >> last;
}

void LASTBIL::skriv()
{
   FORDON::skriv();
   cout << last << endl;
}


// huvudprogram -- lastbilm.cpp

#include "lastbil.h"

void main()
{

   LASTBIL minlb;

   minlb.las();
   minlb.skriv();
}
```

Ett körexempel :

Regnr : ERT123
Ägare : Leif Andersson
Lastvikt : 10

ERT 123
Leif Andersson
10

1.2.3 Dynamisk bindning

Dynamisk bindning innebär att programmet *under körning kan hitta rätt operation* även om man använder pekare till ett basklass-objekt som utgångspunkt och till denna pekare tilldelar en adress för ett subklass-objekt.

Ex : Använder vi ett huvudprogram för lastbilarna ovan enligt :

```
// huvudprogram -- lastbilm.cpp

#include "lastbil.h"

void main()
{

   LASTBIL minlb;
   FORDON *minford;

   minlb.las();
   minlb.skriv();

   minford = &minlb;
   minford->skriv();
};
```

kan ett körexempel se ut som :

Regnr : ERT123
Ägare : Leif Andersson
Lastvikt : 10

ERT 123
Leif Andersson
10

ERT123
Leif Andersson

Den sista utskriften ger *enbart utskrift av ett fordon-objekt*. Programmet kan alltså inte dynamiskt hitta rätt funktion för utskrift, utan utnyttjar den funktion som gäller för den utpekade basklassen. I C++ råder man bot på detta genom att man skriver *virtual* framför den funktion i basklassens specifikation, som ska bindas dynamiskt.

Ändrar man specifikationen av klassen fordon enligt :

```
class FORDON
{
   private:
      char regnr[10];
      char owner[40];
            public:
      void las();
      virtual void skriv();         // OBS! virtual
};
```

kommer utskriften istället att bli :

Regnr : ERT123
Ägare : Leif Andersson
Lastvikt : 10

ERT 123
Leif Andersson
10

ERT123
Leif Andersson
10

OBS! Markeringen *virtual* framför funktionen skriv i fordon-specifikationen. Vill man att även las ska hittas dynamiskt måste även den markeras *virtual*.

OBS! Man kan tilldela *ett subklass-objekt till ett basklass-objekt men inte tvärtom*. Alla lastbilar ovan är alltså fordon och kan tilldelas fordon eller fordon-pekare kan tilldelas adresser för lastbilar. Däremot kan man inte tilldela en lastbil ett fordon. Jfr med att man beställer ett fordon. Då måste man acceptera att få sig tilldelat vilket fordon som helst lastbil, personbil, mc etc. Om man däremot beställer en lastbil ska man inte acceptera att få sig tilldelat en mc, alltså ett fordon.