Collections

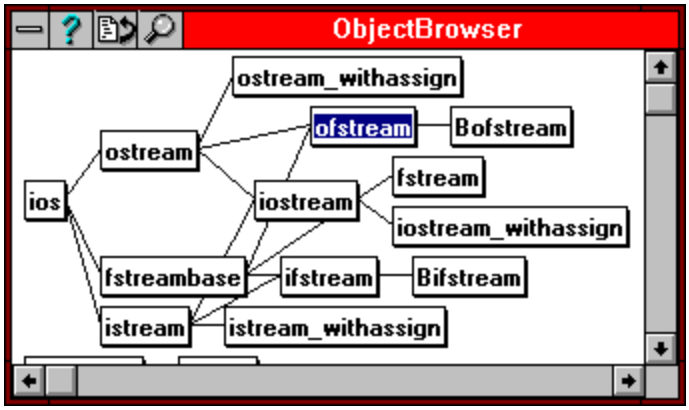
inför på detta sätt klassen Cykel2.
Hur får man in en Bil i en container? Definition av klassen Bil2
Hur blir en Bil en FordonLink? Jo t.ex. genom multipelt arv. Vi inför en koppling mellan dessa klasser med hjälp av multipelt arv. Vi inför på detta sätt klassen Bil2.
Definition av klassen FordonsContainer
Klassen FordonError
Användning av Fordon, Bil och Cykel

Övningar

Övning

"Envisa" objekt

Antag att vi vill att våra objekt på ett enkelt och enhetligt sätt skall bli det som brukar kallas för "**persistant**". Detta innebär att de på ett smidigt sätt skrivs ner och hämtas från en **stream.** Eftersom våra objekt skall kunna innehålla godtyckliga typer av data inför vi klassen **Bofstream** för lagring av objekt och **Bifstream** för att läsa in objekt. Vi låter dessa klasser vara specialfall av **istream** och **ostream** i streamsbiblioteket.



Klasserna Bofstream och Bifstream

Vi inför klasserna på ett enkelt sätt genom att bara införa en konstruktor i respektive klass samt ärva från **ifstream** och **ofstream**.

Vi antar för enkelhetens skull att våra objekt endast kommer att ha data som är av typen **Dvalue , Ivalue** eller **Streng**.

Dvalue och **Ivalue** definieras med typedef-deklarationer:

```
// adef.h
typedef int Ivalue ;
typedef double Dvalue;
```

Klassen **Streng** är en klassisk strängklass som vi skall visa senare.

Vi överlagrar << och >> för objekt av dessa typer.

```
#ifndef BFSTREAM_H
#define BFSTREAM_H
#include <fstream.h>
#include "streng.h"
#include "adef.h"
class Bofstream : public ofstream {
public:
```

```
        Bofstream(Streng);
};
```

```
class Bifstream : public ifstream {
public:
       Bifstream(Streng);
};
Bofstream& operator << (Bofstream& bos, Streng& s);
Bofstream& operator << (Bofstream& bos, Dvalue& v);
Bofstream& operator << (Bofstream& bos, Ivalue& i);

Bifstream& operator >> (Bifstream& bif, Streng& s);
Bifstream& operator >> (Bifstream& bif, Dvalue& v);
Bifstream& operator >> (Bifstream& bif, Ivalue& i);
#endif
```

Definition av metoder i klasserna Bifstream och Bofstream

```
#include <string.h>
#include "adef.h"
#include "streng.h"
#include "bfstream.h"

Bofstream::Bofstream(Streng file)
:ofstream(file,ios::out|ios::binary)
{
}

Bofstream& operator << (Bofstream& bos, Streng& s)
{
      int size = strlen(s);
      bos.write((unsigned char*)&size,sizeof(size));
      bos.write(s,size+1);
      return bos;
}

Bofstream& operator << (Bofstream& bos, Dvalue& v)
{
      bos.write((const unsigned char*)&v,sizeof(v));
      return bos;
}

Bofstream& operator << (Bofstream& bos, Ivalue& i)
{
      bos.write((const unsigned char*)&i,sizeof(i));
      return bos;
}
```

```
Bifstream::Bifstream(Streng file)
:ifstream(file,ios::in|ios::binary)
{
}
Bifstream& operator >> (Bifstream& bif, Streng& s)
{
      int size;
      bif.read((unsigned char*)&size,sizeof(size));
      char *buffer = new char[size+1];
      bif.read(buffer,size+1);
      s = buffer;
      delete [] buffer;
      return bif;
}
Bifstream& operator >> (Bifstream& bif, Dvalue& v)
{
      bif.read(( unsigned char*)&v,sizeof(v));
      return bif;
}
Bifstream& operator >> (Bifstream& bif, Ivalue& i)
{
      bif.read((unsigned char*)&i,sizeof(i));
      return bif;
}
```

Klassen Streng

Som synes är **Streng** en "klassisk" liten strängklass.

```
// streng.h
#ifndef STRENG_H
#define STRENG_H
class Streng {
public:
     Streng(char *p);
     Streng(const Streng&);
     ~Streng();
     Streng& operator=(const Streng&);
     operator const char*() const;
private:
     char *buffer;
     static int ant;
public:
     static int antal() { return ant;}
};
#endif

// streng.cpp
#include <string.h>
#include "streng.h"
int Streng::ant;
Streng::Streng(char *str)
: buffer(strcpy(new char[strlen(str)+1],str))
{
     ant++;
}
Streng::Streng(const Streng& strref)
: buffer(strcpy(new char[strlen(strref.buffer)+
1],strref.buffer))
{
     ant++;
}
Streng::~Streng()
{
     delete [] buffer;
     ant--;
}
```

```
Streng& Streng::operator=(const Streng& strref)
{
     if (this != &strref)
    {
        delete [] buffer;
        buffer =
        strcpy(new char[strlen(strref.buffer)+1],
               strref.buffer);
     }
     return *this;
}
Streng::operator const char *() const
{
     return buffer;
}
```

Klasser som skall bli "envisa"

Om våra övriga applikationsklasser skall bli "**persistant**" måste dessa följa ett viss protokoll:

1. De måste definiera in- och utmatning från **Bifstream** och **Bofstream**. dvs de måste definera vad av klassens datamedlemmar som skall lagras.

2 De får bara lagra data som går att skriva ned på en **Bofstream** (Dvalue, Ivalue och Streng)

Klassen Anstelld skall bli persistant

```
#ifndef ANSTELLD_H
#define ANSTELLD_H
#include "bfstream.h"
#include "adef.h"
class Anstelld {
friend Bofstream& operator<<(Bofstream& bos,Anstelld& a);
friend Bifstream& operator>>(Bifstream& bif,Anstelld& a);
public:
      Anstelld():nr(0),namn(""),adress(""),lon(0) {}
      Anstelld(Ivalue nr,
               Streng namn,
               Streng adress,
               Dvalue lon);
      Anstelld(const Anstelld&);
      ~Anstelld() {}
      Anstelld& operator=(const Anstelld&);
      int operator < (const Anstelld&);
      void print();
private:
      Ivalue nr;
      Streng namn;
      Streng adress;
      Dvalue lon;
};
#endif
```

Definition av medlemmar i klassen Anstelld

Så här har vi definerat klassens medlemmar och vänner:

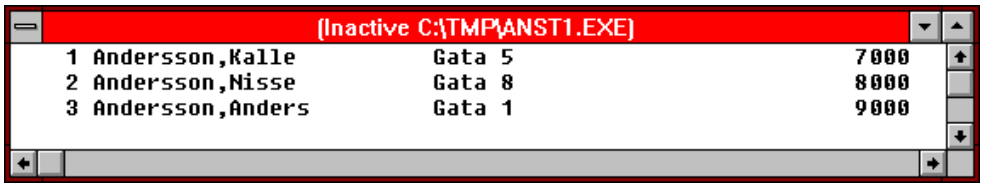
```
#include <string.h>
#include <iomanip.h>
#include "streng.h"
#include "anstelld.h"
Anstelld::Anstelld(Ivalue _nr,Streng _namn,Streng _adress,Dvalue _lon)
:nr(_nr),namn(_namn),adress(_adress),lon(_lon)
{}

void Anstelld::print()
{
      cout << setw(5)  <<  nr << setw(25) << namn
           << setw(25) << adress << setw(20) << lon << endl;

}

int Anstelld::operator < (const Anstelld& a)
{
      return strcmp(namn,a.namn)< 0;
}

Bofstream& operator<<(Bofstream& bos,Anstelld& a)
{
      bos << a.nr;
      bos << a.namn;
      bos << a.adress;
      bos << a.lon;
        return bos;
}
Bifstream& operator>>(Bifstream& bif,Anstelld& a)
{
      bif >> a.nr;
      bif >> a.namn;
      bif >> a.adress;
      bif >> a.lon;
        return bif;
}
```

Användning av Anstelld, Bifstream och Bofstream

Nu kan vi använda klassen **Anstelld** och genom att vi gjort klassen **persistant** kan vi på ett relativt enkelt sätt skriva ner och läsa våra objekt.

```
// anst1.cpp
#include "bfstream.h"
#include "anstelld.h"

void funk1()
{
      // Skapa tre objekt
      Anstelld a(1,"Andersson,Kalle","Gata 5",7000),
             b(2,"Andersson,Nisse","Gata 8",8000),
             c(3,"Andersson,Anders" ,"Gata 1",9000);

        // Skriv ner på fil....
      Bofstream  ut("data.dat");
      ut << a << b << c;
      ut.close();
}

void funk2()
{
        Anstelld a;
      // Läs in från fil...
      Bifstream in("data.dat");
      if (in)
      {
            in >> a;
            while (in)
            {
                  a.print();
                  in >> a;
            }
            in.close();
      }
}

main()
{
      funk1();
      funk2();
      return 0;
}
```

Resultat


(Inactive C:\TMP\ANST1.EXE)
1 Andersson,Kalle Gata 5 7000
2 Andersson,Nisse Gata 8 8000
3 Andersson,Anders Gata 1 9000

En containerklass som är persistant och integrerad med Anstelld

Vi vill kunna lagra ett godtyckligt antal objekt av typen **Anstelld**i någon typ av dynamisk datastruktur. Vi väljer en länkad lista och inför därför klasserna **`AnstellListItem, AnstellList`** samt **`AnstellListIterator.`**

Eftersom vi på ett enkelt sätt vill kunna läsa in och skriva objekt som ligger i en **`AnstellList`** från och till en **`Bifstream`** respektive **`Bofstream`** gör vi **`AnstellList`** persistant.

```
// alist.h
#include "anstelld.h"
class AnstellListItem {
      friend class AnstellList;
public:
      AnstellListItem();
      AnstellListItem(Anstelld&,AnstellListItem *root );
      AnstellListItem(const AnstellListItem&);
      AnstellListItem& operator= (const AnstellListItem&);
        ~AnstellListItem();
      AnstellListItem *getnext();
      Anstelld *getitem();
      void setnext(AnstellListItem *);
private:
      AnstellListItem *next;
      Anstelld *item;
};

class AnstellList {
      friend Bofstream& operator<<(Bofstream& bos,
                                   List& a);
      friend Bifstream& operator>>(Bifstream& bif,
                                   List& a);
      friend Bifstream& operator>>(Bifstream& bif,
                                   List* &lptr);
      friend class AnstellListIterator;
public:
      AnstellList();
      AnstellList(const AnstellList&);
      ~AnstellList();
      AnstellList& operator= (const AnstellList&);
      void sort();
      void add(Anstelld&);
      void ForEach(void (Anstelld::*mptr)());
private:
      AnstellListItem *root;
};

class AnstellListIterator {
```

```
public:
      AnstellListIterator(AnstellList& li);
      operator const void *();
      Anstelld * operator++(int);
private:
      AnstellListItem *current;
};
```

Klassen AnstellListItem

```
#include "alist.h"
AnstellListItem::AnstellListItem(Anstelld& a,
                                 AnstellListItem *root )
: next(root),item(&a)
{
}
AnstellListItem::~AnstellListItem()
{
}
AnstellListItem *AnstellListItem::getnext()
{
      return next;
}

Anstelld *AnstellListItem::getitem()
{
      return item;
}
void AnstellListItem::setnext(AnstellListItem *_next)
{
      next = _next;
}
```

Klassen AnstellList

```
AnstellList::AnstellList()
:root(0)
{
}

AnstellList::AnstellList(const AnstellList& li)
:root(0)
{
      AnstellListItem *item = li.root;
      while (item)
      {
            root = new AnstellListItem(
                              *item->getitem(),root);
            item = item->getnext();
```

```
	}
}

AnstellList::~AnstellList()
{
	AnstellListItem *item = root;
	while (item)
	{
		AnstellListItem *tmp = item;
		item = item->next;
		delete tmp;
	}
}
```

```
AnstellList& AnstellList::operator=(const AnstellList& li)
{

      AnstellListItem *old = root; // Spara gamla listan
      AnstellListItem *item = li.root; //

      root = 0;
      while (item)
      {
            root = new AnstellListItem(
                              *item->getitem(),root);
            item = item->getnext();
      }


      while (old)
      {
            AnstellListItem *tmp = old;
            old = old->next;
            delete tmp;
      }

      return *this;
}
void AnstellList::add(Anstelld& a)
{
      root = new AnstellListItem(a,root);
}
```

Vi inför en metod som sorterar listan genom att traversera listan och skapa en sorterad kopia och till sist radera den osorterade listan.

```
void AnstellList::sort()
{
      if (!root)
            return;

      AnstellListItem *newroot  = new AnstellListItem(
                                          *root->getitem(),0);
      AnstellListItem *tmpnew,*tmporg,*prev;

      tmporg = root->getnext(); // peka på gamla

      // Så länge det finns fler....
      while (tmporg)
        {
              // Peka ut nya....
            tmpnew = newroot;
              prev   = 0;
              // Sök lämplig plats
            while ( tmpnew && *tmpnew->getitem() <
                    *tmporg->getitem())
            {
                  // kom ihåg föregående....
                  prev = tmpnew;
                  // Peka på nästa...
                  tmpnew = tmpnew->getnext();
            }

            if (prev==0)
                  newroot = new AnstellListItem(
                              *tmporg->getitem(),newroot);
            else
                  prev->setnext(
                              new AnstellListItem(
                                          *tmporg->getitem(),
                                           tmpnew)
                              );
            tmporg = tmporg->getnext();
        }
      AnstellListItem *item = root;
      while (item)
      {
            AnstellListItem *tmp = item;
            item = item->next;
              delete tmp;
      }

        root = newroot;
}
```

Funktionen ForEach

Funktionen **ForEach** tar en pekare till en funktion i klassen **Anstelld** och anropar den metoden för varje objekt i listan.

```
void AnstellList::ForEach(void (Anstelld::*mptr)())
{
      AnstellListIterator iter = *this;
      while (iter)
            (iter++->*mptr)();
}
```

Lägg märke till satsen:

```
            (iter++->*mptr)();
```

Vi itererar och får en pekare till nästa objekt i listan. För detta objekt anropar vi medlemsfunktionen som mptr pekar på. Denna måste vara av typen void och tar inga parametrar.

Att skriva ner och läsa ett objekt av typen AnstellList

Vi inför utskrift och inmatning av referens till ett objekt av typen **AnstellList.**

```
Bofstream& operator<<(Bofstream& bos,AnstellList& a)
{
      AnstellListItem *item = a.root;
      while (item)
      {
            bos << *item->getitem();
            item = item->getnext();
        }

      return bos;
}

Bifstream& operator>>(Bifstream& bif,AnstellList& a)
{
      Anstelld *aptr = new Anstelld;
      while (bif >> *aptr)
        {
            a.add(*aptr);
            aptr = new Anstelld;
      }
        delete aptr;

        return bif;
}
```

Sedan inför vi också inläsning av **AnstellList** från **Bifstream** till referens till pekare till **AnstellList.** När vi läser in listan skapar funktionen automatiskt ett listobjekt och fyller detta med objekt av typen **Anstelld** från en ström. Genom att vi använder en referens till en pekare kommer vi indirekt att manipulera med den pekare som används vid anropet.

```
Bifstream& operator>>(Bifstream& bif,AnstellList*& lptr)
{
      lptr = new AnstellList;
      Anstelld *aptr = new Anstelld;
      while (bif >> *aptr)
      {
            lptr->add(*aptr);
            aptr = new Anstelld;
      }
      delete aptr;

      return bif;
}
```

Klassen AnstellListIterator

Sedan definerar vi en iteratorklass som kan användas för att iterera över alla objekt i ett objekt av typ **AnstellList.**

```
AnstellListIterator::AnstellListIterator(AnstellList& li)
:current(li.root)
{
}

Anstelld* AnstellListIterator::operator++(int)
{
      AnstellListItem *item = current;
      if (item)
            current = current->getnext();
        return item->getitem();;
}

AnstellListIterator::operator const void *()
{
      return current;
}
```

Användning av containerklassen

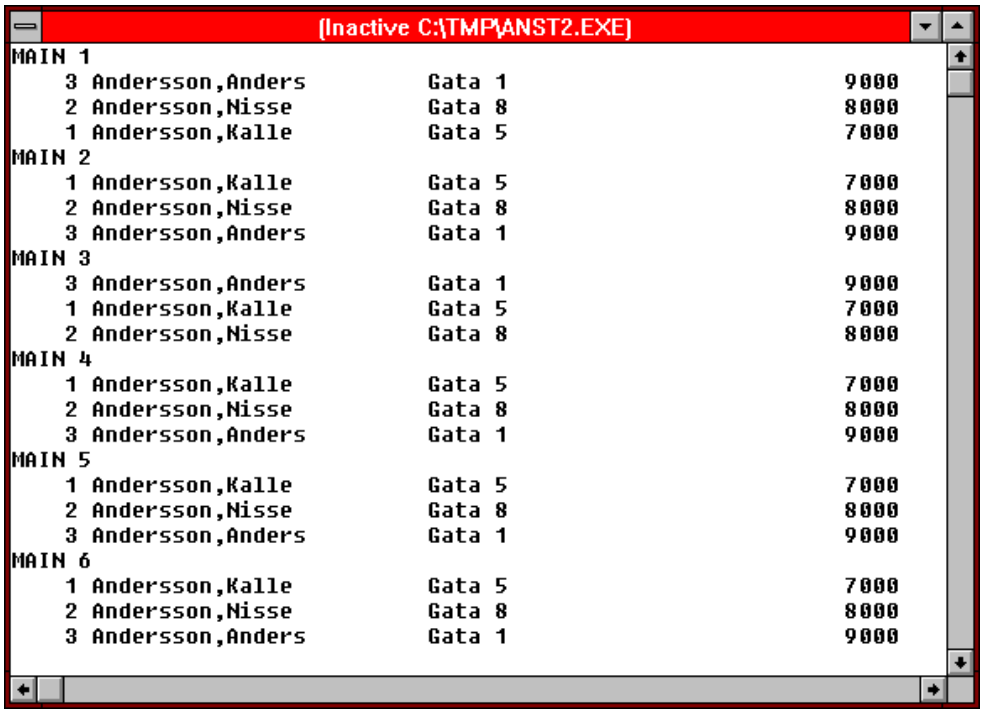
Vi prövar nu våra klasser och de funktioner vi definierat:

```
// anst2.cpp
#include "anstelld.h"
#include "alist.h"

main()
{
      // Skapa tre objekt
      Anstelld a(1,"Andersson,Kalle","Gata 5",7000),
             b(2,"Andersson,Nisse","Gata 8",8000),
             c(3,"Andersson,Anders" ,"Gata 1",9000);

      // En lista....
      AnstellList lista1;

      // Lägg in i listan....
      lista1.add(a);
      lista1.add(b);
      lista1.add(c);

      // Skriv ner på fil....
      Bofstream  ut("data.dat");
      ut << lista1;
      ut.close();

      cout << "MAIN 1\n";
      // Iterera....
      AnstellListIterator iter = lista1;
      while (iter)
            (iter++)->print();

      // Ett listobjekt....
      AnstellList lista2;

      // Läs in från fil....
      Bifstream in("data.dat");
      in >> lista2;
      in.close();

      cout << "MAIN 2\n";
      // Iterera....
      iter = lista2;
      while (iter)
            (iter++)->print();

      lista2.sort();

      cout << "MAIN 3\n";
```

```
// Iterera....
iter = lista2;
while (iter)
      (iter++)->print();
```

```
	cout << "MAIN 4\n";
	AnstellList lista3 = lista1;
	  // Iterera....
	iter = lista3;
	while (iter)
		(iter++)->print();

	cout << "MAIN 5\n";
	lista3 = lista1;
	lista3.ForEach(&Anstelld::print);

	AnstellList *lista4;

	// Läs in från fil....
	  Bifstream in2("data.dat");
	  in2 >> lista4;
	in2.close();
	cout << "MAIN 6\n";
	lista4->ForEach(&Anstelld::print);

	// Iterera....
	iter = *lista4;
	while (iter)
		delete iter++;

	delete lista4;

	return 0;
}
```

Resultat


(Inactive C:\TMP\ANST2.EXE)
MAIN 1
3 Andersson,Anders Gata 1 9000
2 Andersson,Nisse Gata 8 8000
1 Andersson,Kalle Gata 5 7000
MAIN 2
1 Andersson,Kalle Gata 5 7000
2 Andersson,Nisse Gata 8 8000
3 Andersson,Anders Gata 1 9000
MAIN 3
3 Andersson,Anders Gata 1 9000
1 Andersson,Kalle Gata 5 7000
2 Andersson,Nisse Gata 8 8000
MAIN 4
1 Andersson,Kalle Gata 5 7000
2 Andersson,Nisse Gata 8 8000
3 Andersson,Anders Gata 1 9000
MAIN 5
1 Andersson,Kalle Gata 5 7000
2 Andersson,Nisse Gata 8 8000
3 Andersson,Anders Gata 1 9000
MAIN 6
1 Andersson,Kalle Gata 5 7000
2 Andersson,Nisse Gata 8 8000
3 Andersson,Anders Gata 1 9000

En templatebaserad containerklass

Vi skall nu visa hur vi kan gå vidare och konstruera en containerklass som är persistant och kan lagra objekt av godtycklig typ som också är persistant.

Vi vill kunna lagra ett godtyckligt antal objekt av en **godtycklig typ** i en **dynamisk** datastruktur. Vi väljer en länkad lista och inför därför templateklasserna **`ListItem<Type>, List<Type>`** samt **`ListIterator<Type>.`**

Eftersom vi på ett enkelt sätt vill kunna läsa in och skriva objekt som ligger i en från och till en Bifstream respektive Bofstream gör vi vår templatebaserade listklass persistant.

Klassdefinitioner för våra nya templateklasser

Klassen ListItem<Type>

```
#ifndef ALIST2_H
#define ALIST2_H
template <class Type>
class ListItem {
      friend class List;
public:
      ListItem();
      ListItem(Type&, ListItem *root );
      ListItem(const ListItem&);
      ListItem<Type>& operator= (const ListItem&);
      ~ListItem();
      ListItem<Type> *getnext();
      Type *getitem();
      void setnext(ListItem *);
private:
      ListItem<Type> *next;
      Type *item;
};
```

Klassen List<Type>

```
template <class Type>
class List {
  friend
  Bofstream& operator<<(Bofstream& bos,List<Type>& a);
  friend
  Bifstream& operator>>(Bifstream& bif,List<Type>& a);
  friend
  Bifstream& operator>>(Bifstream& bif,List<Type>* &lptr);
  friend
  class ListIterator<Type>;
public:
      List();
      List(const List&);
      ~List();
      List& operator= (const List&);
      void sort();
      void add(Type&);
      void ForEach(void (Type::*mptr)());
private:
      ListItem<Type> *root;
};
```

Klassen ListIterator<Type>

```
template <class Type>
class ListIterator {
public:
      ListIterator(List<Type>& li);
        operator const void *();
      Type * operator++(int);
private:
      ListItem<Type> *current;
};
#include "alist2.icc"
#endif
```

Funktionsdefinitioner för våra templatefunktioner och klasser

```
// alist2.icc
#include "bfstream.h"
template <class Type>
ListItem<Type>::ListItem(Type& a, ListItem<Type> *root )
: next(root),item(&a)
{
}
template <class Type>
ListItem<Type>::~ListItem()
{
}
template <class Type>
ListItem<Type>* ListItem<Type>::getnext()
{
      return next;
}

template <class Type>
Type *ListItem<Type>::getitem()
{
      return item;
}
template <class Type>
void ListItem<Type>::setnext(ListItem<Type> *_next)
{
      next = _next;
}

template <class Type>
List<Type>::List()
:root(0)
{
}

template <class Type>
List<Type>::List(const List<Type>& li)
:root(0)
{
      ListItem<Type> *item = li.root;
      while (item)
      {
            root = new ListItem<Type>(*item->getitem(),root);
            item = item->getnext();
      }
}
```

```
template <class Type>
List<Type>::~List()
{
      ListItem<Type> *item = root;
      while (item)
      {
            ListItem<Type> *tmp = item;
            item = item->getnext();
                delete tmp;
      }
}
template <class Type>
List<Type>& List<Type>::operator=(const List<Type>& li)
{

      ListItem<Type> *old = root; // Spara gamla listan
      ListItem<Type> *item = li.root; //

        root = 0;
      while (item)
      {
            root = new ListItem<Type>(*item->getitem(),root);
            item = item->getnext();
      }
      while (old)
      {
            ListItem<Type> *tmp = old;
            old = old->getnext();
            delete tmp;
      }
      return *this;
}
template <class Type>
void List<Type>::add(Type& a)
{
      root = new ListItem<Type>(a,root);
}
```

```
template <class Type>
void List<Type>::sort()
{
      if (!root)
            return;

      ListItem<Type> *newroot  = new
                        ListItem<Type>(*root->getitem(),0);

      ListItem<Type> *tmpnew;
      ListItem<Type> *tmporg;
      ListItem<Type> *prev;

      tmporg = root->getnext(); // peka på gamla

      // Så länge det finns fler....
      while (tmporg)
      {
            // Peka ut nya....
            tmpnew = newroot;
            prev   = 0;
            // Sök lämplig plats
            while ( tmpnew && *tmpnew->getitem() <
                      *tmporg->getitem())
            {
                  // kom ihåg föregående....
                  prev = tmpnew;
                  // Peka på nästa...
                  tmpnew = tmpnew->getnext();
            }

            if (prev== 0)
            {
                  newroot = new  ListItem<Type>(
                                    *tmporg->getitem(),
                                    newroot);
            }
            else
            {
                  prev->setnext(
                        new ListItem<Type>(*tmporg->getitem(),
                        tmpnew));
            }
            tmporg = tmporg->getnext();
      }
      ListItem<Type> *item = root;
      while (item)
      {
            ListItem<Type> *tmp = item;
            item = item->getnext();
                delete tmp;
      }

      root = newroot;
```

```
}
```

List<Type>::ForEach::void (Type::*mptr)())

```
template <class Type>
void List<Type>::ForEach(void (Type::*mptr)())
{
      ListIterator<Type> iter = *this;
      while (iter)
            (iter++->*mptr)();
}
```

Templatefunktioner för in och utmatning av List<Type>

```
template <class Type>
Bofstream& operator<<(Bofstream& bos,List<Type>&a)
{
      ListItem<Type>*item = a.root;
      while (item)
      {
            bos << *item->getitem();
            item = item->getnext();
        }

      return bos;
}
template <class Type>
Bifstream& operator>>(Bifstream& bif,List<Type>& a)
{
      Type *aptr = new Type;
      while (bif >> *aptr)
        {
            a.add(*aptr);
            aptr = new Type;
      }
        delete aptr;

        return bif;
}
template <class Type>
Bifstream& operator>>(Bifstream& bif,List<Type>* &lptr)
{
      lptr = new List<Type>;
      Type *aptr = new Type;
      while (bif >> *aptr)
      {
            lptr->add(*aptr);
            aptr = new Type;
      }
        delete aptr;

        return bif;
```

```
}
```

ListIterator<Type>

```
template <class Type>
ListIterator<Type>::ListIterator(List<Type>& li)
:current(li.root)
{
}
template <class Type>
Type* ListIterator<Type>::operator++(int)
{
      ListItem<Type> *item = current;
      if (item)
            current = current->getnext();
        return item->getitem();
}
template <class Type>
ListIterator<Type>::operator const void *()
{
      return current;
}
```

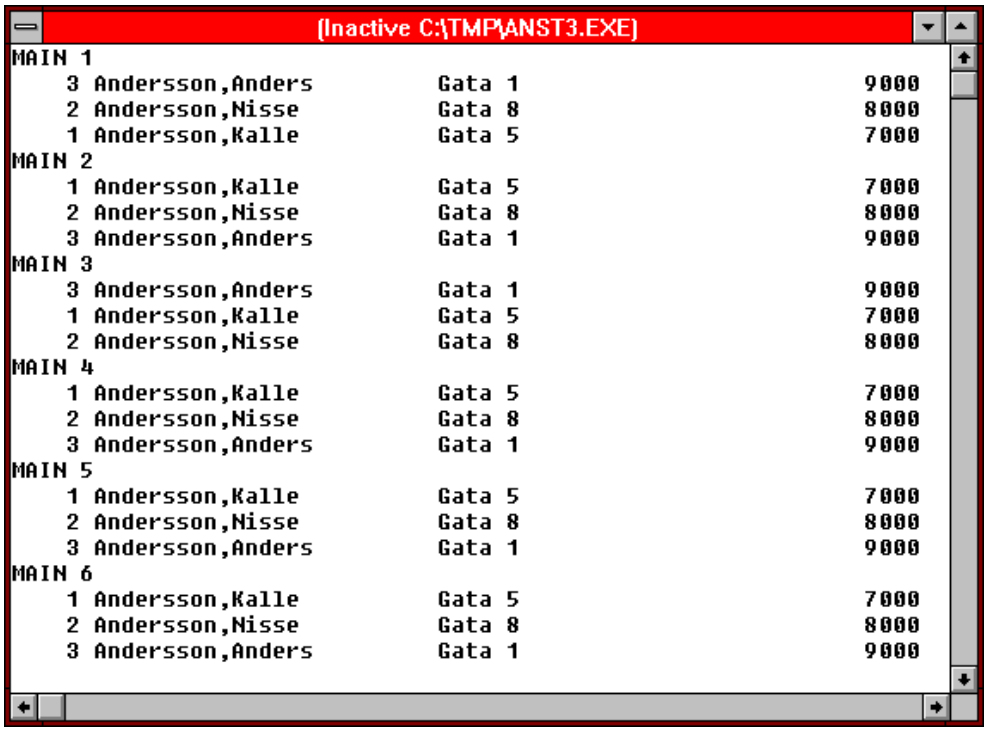
Användning av våra templateklasser

```
#include "anstelld.h"
#include "alist2.h"

main()
{
      // Skapa tre objekt
      Anstelld    a(1,"Andersson,Kalle","Gata 5",7000),
                  b(2,"Andersson,Nisse","Gata 8",8000),
                  c(3,"Andersson,Anders" ,"Gata 1",9000);

      // En lista....
      List<Anstelld> lista1;

      // Lägg in i listan....
      lista1.add(a);
      lista1.add(b);
      lista1.add(c);

      // Skriv ner på fil....
      Bofstream  ut("data.dat");
      ut << lista1;
      ut.close();

      cout << "MAIN 1\n";
      // Iterera....
      ListIterator<Anstelld> iter = lista1;
      while (iter)
            (iter++)->print();
```

```
// Ett listobjekt....
List<Anstelld> lista2;

// Läs in från fil....
Bifstream in("data.dat");
in >> lista2;
in.close();

cout << "MAIN 2\n";
// Iterera....
iter = lista2;
while (iter)
      (iter++)->print();

lista2.sort();

cout << "MAIN 3\n";
// Iterera....
iter = lista2;
while (iter)
      (iter++)->print();

// Iterera....
iter = lista2;
while (iter)
      delete iter++;

cout << "MAIN 4\n";
List<Anstelld> lista3 = lista1;

// Iterera....
iter = lista3;
while (iter)
      (iter++)->print();

cout << "MAIN 5\n";
lista3 = lista1;
lista3.ForEach(&Anstelld::print);

List<Anstelld> *lista4;

// Läs in från fil....
Bifstream in2("data.dat");
in2 >> lista4;
in2.close();

cout << "MAIN 6\n";
lista4->ForEach(&Anstelld::print);
// Iterera....
iter = *lista4;
while (iter)
      delete iter++;
delete lista4;
```

```
      return 0;
}
```

Resultat



Heterogena kollektioner

Ett kärt ämne bland C++-programmerare är att diskutera hur man på ett smart sätt löser problemet med att lagra olika typer av objekt i samma datastruktur och att sedan kunna plocka fram dem och automatiskt betrakta dem som av rätt typ.

Lösningar på detta problem brukar vanligtvis bestå av någon form av arvsmekanismer och polymorfism. Vi skall nu gå igenom ett exempel som bygger på att vi från början har förutsättningar som vi inte kan, får eller vill ändra på.

Antag att vi sedan tidigare har konstruerat och använt två klasser oberoende av varandra och att vi behöver kunna lagra dessa i **ett** containerobjekt av något slag. Problemet är att vi **inte** tillåter att vi inför en gemensam basklass. Förutsättningen är ju att vi inte får ändra i de existerande klasserna.

Tidigare i det här avsnittet har vi visat hur man kan skriva en speciell container för objekt av en speciell typ (Anstelld). Vi har också visat hur man kan skriva en templateklass för en container och med vars hjälp man kan få kompilatorn att automatiskt generera klasser för vilken typ som helst som uppfyller vissa kriterier.

Det vore inga större problem att skapa en containerklass för varje typ. Men detta löser inte vårt problem.

Två oberoende klasser som skall in i samma container.

Antag att vi tidigare i samma projekt eller i ett tidigare projekt definerat klasserna **Bil** och **Cykel**. Antag att vi vill skapa en gemensam containerklass för dessa typer av objekt.

Klassen Cykel

Klassen **Cykel** är en abstraktion av begreppet "cykel som är till salu". Ett cykelobjekt vet sitt namn, vilken årsmodell det är samt hur mycket den kostar. En cykel har metoder för tilldelning och initiering samt för erhållande om information. Dessutom vet Cykelklassen hur många objekt av denna typ som för närvarande existerar.

```
#ifndef CYKEL_H
#define CYKEL_H
#include "streng.h"
class Cykel {
public:
       Cykel(Streng namn = "Okänd", long year=0,long
pris=0);
       Cykel(const Cykel&);
       virtual ~Cykel();
       Cykel& operator = (const Cykel&);
       const char* show(void);
private:
       Streng namn;
       long   year;
       long   pris;
       static int ant;
public:
     static int antal() { return ant;}
};
#endif
```

```
#include <iomanip.h>
#include <string.h>
#include <strstream.h>
#include "streng.h"
#include "cykel.h"

int Cykel::ant;

Cykel::Cykel(Streng _namn,long _year,long _pris)
:namn(_namn),year(_year),pris(_pris)
{
     ant++;
}

Cykel::Cykel(const Cykel& cykel)
:namn(cykel.namn),year(cykel.year),pris(cykel.pris)
{
     ant++;
}
```

```
Cykel::~Cykel()
{
      ant--;
}

Cykel& Cykel::operator =(const Cykel& cykel)
{
      namn = cykel.namn;
      year = cykel.year;
      pris = cykel.pris;

        return *this;
}

const char* Cykel::show(void)
{
      static char buffer[100];
      memset(buffer,'\0',100);
      ostrstream utbuffer(buffer,sizeof(buffer));
      utbuffer.setf(ios::left);
      utbuffer << "Cykel    :" <<  setw(15) << namn
             << "Årsmodell:" <<  setw(5) << year
             << "Pris     :" <<  pris << endl;

      return buffer;
}
```

Klassen Bil

Klassen **Bil** är en abstraktion av begreppet "bil som är till salu". Ett bilobjekt vet sitt namn, vilken årsmodell det är, hur många mil den gått samt hur mycket den kostar. En bil har metoder för tilldelning och initiering samt för erhållande av information. Dessutom vet bilklassen hur många objekt av denna typ som för närvarande existerar.

```
#ifndef BIL_H
#define BIL_H
#include "streng.h"
class Bil {
public:
       Bil(Streng namn = "Okänd",long year=0,long mil=0,long pris=0);
       Bil(const Bil&);
       virtual ~Bil();

       Bil& operator =(const Bil&);

       const char* display(void);

       virtual void run();
private:
       Streng namn;
       long year;
```

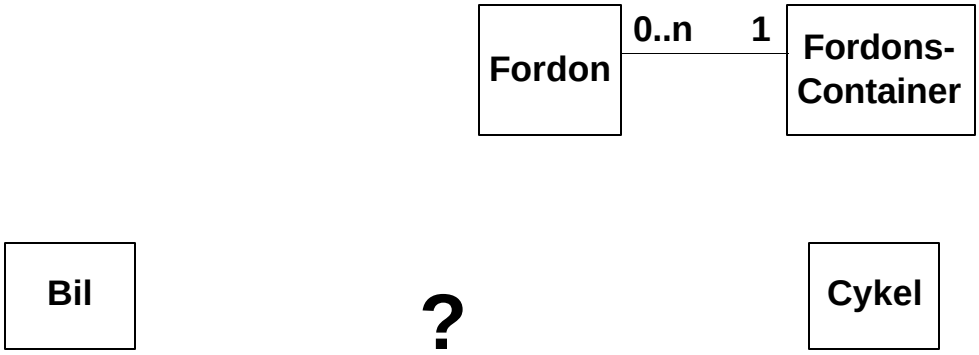
```
        long mil;
        long pris;
        static int ant;
public:
      static int antal() { return ant;}
};
#endif
```

```
// bil.cpp
#include <string.h>
#include <strstream.h>
#include <iomanip.h>
#include "streng.h"
#include "bil.h"
int Bil::ant;
Bil::Bil(Streng _namn,long _year,long _mil,long _pris)
:namn(_namn),year(_year),mil(_mil),pris(_pris)
{
      ant++;
}
Bil::Bil(const Bil& bil)
:namn(bil.namn),year(bil.year),mil(bil.mil),pris(bil.pris)
{
      ant++;
}
void Bil::run()
{
      cout << "Bil is running\n";
}
Bil::~Bil()
{
      ant--;
}
Bil& Bil::operator =(const Bil& bil)
{
      namn = bil.namn;
      year = bil.year;
      mil  = mil;
      pris = pris;

      return *this;
}

const char* Bil::display(void)
{
      static char buffer[100];
      memset(buffer,'\0',100);
      ostrstream utbuffer(buffer,sizeof(buffer));
      utbuffer.setf(ios::left);
      utbuffer << "Bil      :" << setw(15) << namn
            << "Årsmodell:" << setw(5)  << year
            << "Pris     :" << setw(7)  << pris
            << "Mil      :" << mil << endl;

      return buffer;
}
```

En container för Fordon

Eftersom både en **Cykel** och **Bil** passar att betrakta som ett **Fordon** tänker vi försöka att definiera en klass med detta namn och låta denna på något sätt kunna vara eller har en koppling till både en **Bil** och **Cykel**. **Fordon** vill vi sedan lägga i en **FordonsContainer**.

Hur ska vi då göra för att koppla **Fordon** till **Cykel** på ett effektivt sätt?



En omslagsklass (envelope): Fordon

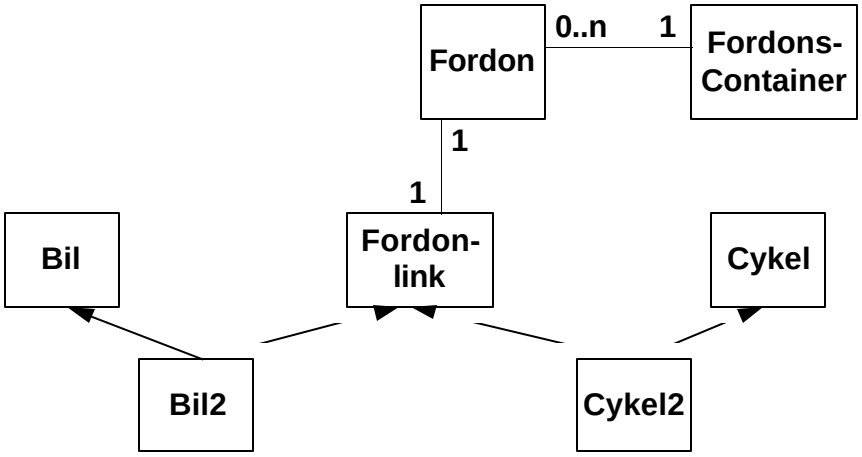
Ett objekt av typen **Fordon** skall kunna initieras, tilldelas och omvandlas till och från en **Bil** eller **Cykel.** Därför inför vi konstruktorer, tilldelnings-operatorer och överlagrad typomvandling mellan dessa typer. Vi inför också metoder för typidentifikation (**IsA**). Varje objekt av typ **Fordon** håller en pekar till ett objekt av typen **FordonLink**.

En hjälpklass: FordonLink

Vi behöver en hjälpklass som länk mellan Fordon och Bil respektive Cykel. Vi kallar denna klass för FordonLink. Klassen får helt virtuella funktioner som skall defineras av klasser som på något sätt skall vara både av typen FordonLink och Bil eller Cykel. FordonLink blir på så sätt en abstrakt klass som påför ett protokoll på den som vill definera en subklass som skall vara en instansklass.

FordonsContainer

Klassen **FordonsContainer** är en traditionell containerklass som skrivs för att hantera en typ av objekt: **Fordon. FordonsContainer** innehåller ingenting som gör den beroende av någon annan klass än **Fordon.** Vi implementerar **FordonContainer** som en säker array med överlagrad indexoperator. Med hjälp av denna kan man lägga in Cyklar och Bilar (letters) genom att dölja dessa under omslaget (envelope) **Fordon.**



Definition av klassen Fordon

```
#ifndef FORDON_H
#define FORDON_H
#include "cykel.h"
#include "bil.h"
enum ClassType { ERROR, BIL, CYKEL };
class FordonError;
class Fordon;
class Fordonlink;
class Fordon {
public:
      Fordon();
      Fordon(const Fordon& fo);
      Fordon(const Bil&);
      Fordon(const Cykel&);
      Fordon(const FordonError&);

      virtual ~Fordon();

      Fordon& operator =(const Fordon& fo);
      Fordon& operator =(const Bil&);
      Fordon& operator =(const Cykel&);
      Fordon& operator =(const FordonError&);

      operator Bil();
      operator Cykel();

      const char* info();
      Fordonlink* operator ->();
      ClassType isA();
private:
        static int ant;
        Fordonlink * pFordonlink;
public:
      static int antal() { return ant;}
};

#endif
```

```
#include <iostream.h>
#include "fordon.h"
#include "forderr.h"
#include "bil2.h"
#include "cykel2.h"
int Fordon::ant;
Fordon::Fordon()
: pFordonlink(new FordonError("Ej initierad!"))
{
      ant++;
}
Fordon::Fordon(const Fordon& fo)
: pFordonlink(fo.pFordonlink->clone())
{
      ant++;
}
Fordon::Fordon(const Bil& bil)
: pFordonlink(new Bil2(bil))
{
      ant++;
}

Fordon::Fordon(const Cykel& cykel)
:  pFordonlink(new Cykel2(cykel))
{
      ant++;
}
Fordon::Fordon(const FordonError& fe)
:pFordonlink(new FordonError(fe))
{
      ant++;
}
Fordon::~Fordon()
{
      delete pFordonlink;
      ant--;
}

Fordon& Fordon::operator =(const Fordon& fo)
{
      delete pFordonlink;
      pFordonlink = fo.pFordonlink->clone();
      return *this;
}
Fordon& Fordon::operator =(const Bil& bil)
{
      delete pFordonlink;
      pFordonlink = new Bil2(bil);
      return *this;
}
Fordon& Fordon::operator =(const Cykel& cykel)
{
      delete pFordonlink;
```

```
        pFordonlink = new Cykel2(cykel);
        return *this;
}
```

```
Fordon& Fordon::operator =(const FordonError& error)
{
      delete pFordonlink;
      pFordonlink = new FordonError(error);
      return *this;
}
Fordon::operator Bil()
{
      if ( pFordonlink->isA() == BIL)
            return *(Bil2*) pFordonlink;
      else
            return  "Ej bil";
}
```

```
Fordon::operator Cykel()
{
      if ( pFordonlink->isA() == CYKEL)
            return *(Cykel2*) pFordonlink;
      else
            return  "Ej cykel";
}
const char* Fordon::info()
{
      return pFordonlink->info();
}
ClassType Fordon::isA()
{
      return pFordonlink->isA();
}
Fordonlink* Fordon::operator ->()
{
      return pFordonlink;
}
```

Definition av klassen FordonLink

Fordonlink är en abstrakt klass som bara kan användas som basklass.

```
class Fordonlink {
      friend class Fordon;
private:
      virtual Fordonlink* clone()  = 0;
      virtual ClassType isA()      = 0;
      virtual Streng info()       = 0;
protected:
      virtual ~Fordonlink() {}
};
#endif
```

Hur blir en Cykel en FordonLink? Definition av klassen Cykel2

Hur blir en Cykel en FordonLink? Jo t.ex. genom multipelt arv. Vi inför en koppling mellan dessa klasser med hjälp av multipelt arv. Vi inför på detta sätt klassen **Cykel2.**

```
#ifndef CYKEL2_H
#define CYKEL2_H
#include "fordon.h"
#include "cykel.h"
class Cykel2 : private Cykel, private Fordonlink
{
     friend class Fordon;
private:
     Cykel2(const Cykel&);
     ~Cykel2();
     ClassType isA();
     Fordonlink* clone();
     const char* info();
     static int ant;
public:
     static int antal() { return ant;}
};
#endif
```

Hur får man in en Bil i en container? Definition av klassen Bil2

Hur blir en Bil en FordonLink? Jo t.ex. genom multipelt arv. Vi inför en koppling mellan dessa klasser med hjälp av multipelt arv. Vi inför på detta sätt klassen **Bil2.**

```
#ifndef BIL2_H
#define BIL2_H
#include "fordon.h"
#include "bil.h"
class Bil2 : private Bil, private Fordonlink {
     friend class Fordon;
private:
     Bil2(const Bil&);
     ~Bil2();
     ClassType isA();
     Fordonlink* clone();
     const char* info();
     static int ant;
public:
     static int antal() { return ant;}
};
#endif
```

Definition av klassen FordonsContainer

```
#ifndef FORDONCO_H
#define FORDONCO_H
class FordonContainer {
public:
     FordonContainer(int size=1);
     FordonContainer(const FordonContainer&);
     ~FordonContainer();
     FordonContainer& operator=(
                         const FordonContainer&);
     Fordon& operator [] (int index);
     const Fordon& operator [] (int index) const;
private:
     Fordon *arr;
     int limit;
};
#endif
```

```
#include <strstrea.h>
#include "fordon.h"
#include "forderr.h"
#include "fordonco.h"
FordonContainer::FordonContainer(int size)
:arr(new Fordon[size]),limit(size)
{
}
FordonContainer::FordonContainer(const FordonContainer& fc)
:arr(new Fordon[fc.limit]),limit(fc.limit)
{
     for(int i=0;i<limit;i++)
          arr[i] = fc.arr[i];
}

FordonContainer::~FordonContainer()
{
     delete [] arr;
}

FordonContainer& FordonContainer::operator=(const FordonContainer& fc)
{
     delete [] arr;
     arr = new Fordon[fc.limit];
     limit = fc.limit;
     for(int i=0;i<limit;i++)
          arr[i] = fc.arr[i];
     return *this;
}
```

```
static Fordon ERRORS;
Fordon& FordonContainer::operator [] (int index)
{
      if (index>=0 && index <limit)
            return arr[index];
      else    {
            static char buffer[100];
            ostrstream error(buffer,sizeof(buffer));
            error << "Fel index:" << index
                  << "(0<index<" << limit << ")";
            ERRORS = FordonError(error.str());
            return ERRORS;
      }
}
const Fordon& FordonContainer::operator [] (int index) const
{
      if (index>=0 && index <limit)
            return arr[index];
      else    {
            static char buffer[100];
            ostrstream error(buffer,sizeof(buffer));
            error << "Fel index:" << index
                  << " (0<index<" << limit << ")";
            ERRORS = FordonError(error.str());
            return ERRORS;
      }
}
```

Klassen FordonError

Vi inför en klass som vi kan använda för felhantering.

```
#ifndef ERROR_H
#define ERROR_H
#include "streng.h"
#include "fordon.h"
class FordonError : private Fordonlink {
      friend class Fordon;
      friend class FordonContainer;

public:
      virtual ~FordonError();
private:
      FordonError(const char*);
      FordonError(const FordonError&);

      ClassType isA();
      Fordonlink* clone();
      const char* info();
      Streng message;
      static int ant;
```

```
public:
      static int antal() { return ant;}
};
#endif

#include <iostream.h>
#include <string.h>
#include <strstream.h>
#include "streng.h"
#include "forderr.h"
int FordonError::ant;
FordonError::FordonError(const char *s)
: message(s)
{
     ant++;
}

FordonError::FordonError(const FordonError& fe)
: message(fe.message)
{
     ant++;
}

FordonError::~FordonError()
{
     ant--;
}


Fordonlink* FordonError::clone()
{
     return new FordonError(*this);
}

ClassType FordonError::isA()
{
     return ERROR;
}
const char* FordonError::info(void)
{
     static char buffer[100];
     memset(buffer,'\0',100);
     ostrstream utbuffer(buffer,sizeof(buffer));
     utbuffer << "Error:" << message << endl;
     return buffer;
}
```

Användning av Fordon, Bil och Cykel

Vi definierar två hjälpfunktioner som vi kan använda för att kontrollera objektidentitet.

```
int isBil(Fordon & fo)
{
     return fo.isA() == BIL;
}

int isCykel(Fordon & fo)
{
     return fo.isA() == CYKEL;
}
```

Vi gör också en dump-funktion så att vi kan kontrollera antalet objekt av respektive typ:

```
void dump()
{
     cout << "Fordon     :" << Fordon::antal() << endl;
     cout << "Bil        :" <<  Bil::antal() << endl;
     cout << "Cykel      :" << Cykel::antal() << endl;
     cout << "Bil2       :" << Bil2::antal() << endl;
     cout << "Cykel2     :" << Cykel2::antal() << endl;
     cout << "FordonError:" << FordonError::antal() << endl;
}
```

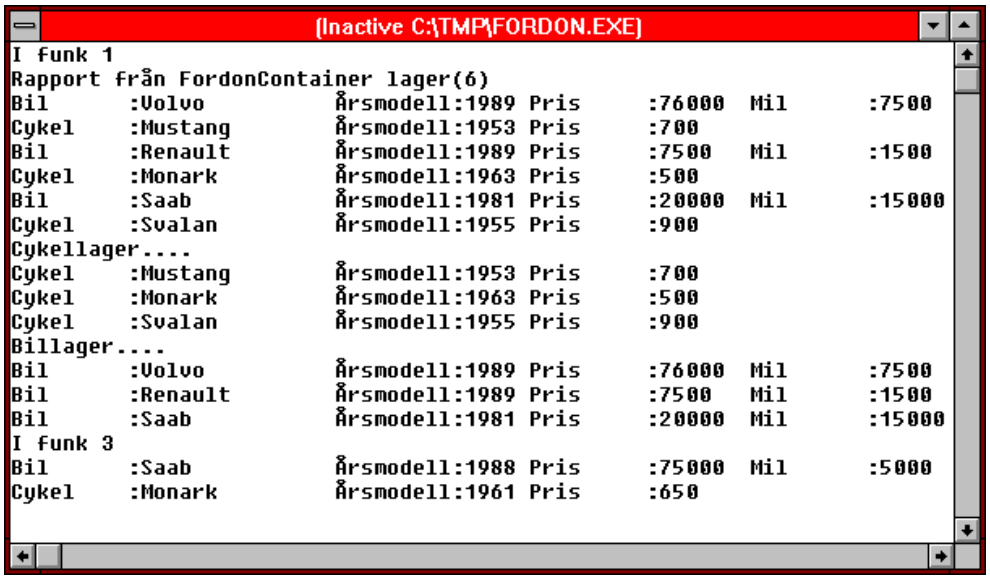
Vi prövar att definiera en funktion som tar ett **Fordon** som parameter. Genom att kontrollera objektet avgör var det är för typ. Vi kan sedan initiera ett objekt av rätt typ och välja rätt funktioner:

```
void funk2(Fordon fordon)
{
     if (isBil(fordon))
     {
          Bil bil = fordon;
          cout << bil.display();
     }
     else
     if (isCykel(fordon))
     {
          Cykel cykel = fordon;
          cout << cykel.show();
     }
}

void funk3()
{
     Bil   b("Saab",1988,5000,75000);
     Cykel c("Monark",1961,650);
     cout << "I funk 3\n";
     funk2(b);
     funk2(c);
}
```

```
void funk()
{
	cout << "I funk 1\n";
	Fordon arr[] = {
			Bil(  "Volvo"  ,1989L , 7500L,76000L),
			Cykel("Mustang",1953L , 700L),
			Bil(  "Renault",1989L , 1500L,7500L),
			Cykel("Monark" ,1963L , 500L),
			Bil(  "Saab"   ,1981L , 15000L,20000L),
			Cykel("Svalan" ,1955L , 900L) };

	FordonContainer lager(6);
	for(int i=0;i<6;i++)
	{
		lager[i] = arr[i];
	}

	cout  << "Rapport från FordonContainer lager(6)"
		<< endl;
	for(i=0;i<6;i++)
	{
		cout << lager[i].info();
	}

	FordonContainer cyklar(6);
	FordonContainer bilar(6);

	int cindex=0;
	int bindex=0;

	for(i=0;i<6;i++)
	{
		if (isCykel(lager[i]))
		{
			cyklar[cindex++] = lager[i];
		}
		else if (isBil(lager[i]))
		{
			bilar[bindex++] = lager[i];
		}
	}
	cout << "Cykellager....\n";
	for(i=0;i<cindex;i++)
	{
		cout << cyklar[i].info();
	}
	cout << "Billager....\n";
	for(i=0;i<bindex;i++)
	{
		cout << bilar[i].info();
	}
}
```

Sedan anropar vi funk och funk3 och utvärderar resultatet:

```
main()
{
     funk();
     funk3();
     return 0;
}
```



Övningar

Övning

Gör en **Containerklass** för en '**heterogen kollektion".** Ett containerobjekt skall hålla ett godtyckligt antal objekt av typen **Base**. **Base** skall vara en abstrakt klass (minst en helt virtuell funktion). Du definierar sedan minst två klasser som använder **Base** som basklass; **Derived1** och **Derived2.** Du skall kunna placera objekt av typerna **Derived1** och **Derived2** i containern och sedan kunna fiska upp dom igen. Genom att kontrollera typen på ett objekt av typ **Base** från containern skall du kunna avgöra fvilka omvandlingar som du kan göra och vilka metoder som i så fall kan användas (de i klassen **Derived1** eller **Derived2).**