Initiering och tilldelning

Övning 1
Övning 2
Övning 3
Övning 4

Datatyper

I C++ kan man göra en grov uppdelning av de datatyper som vi använder:

- Inbyggda datatyper
- Abstrakta datatyper
- Konkreta datatyper

Inbyggda datatyper

T.ex. int, float, long och double. Förutom vissa implementationsberoende egenskaper finns för dessa ett väldefinierat regelsystem som gäller oberoende av vilken kompilator och vilket operativsystem vi använder.

Exempel på detta är t.ex. hur de fungerar ihop med olika operatorer, hur de skall överföras till och från funktioner och vilka tilldelningar och typomvandlingar som är möjliga. Man kan t.ex. betrakta de inbyggda typerna som en abstraktion som stödjer den logik och de matematiska begrepp som ofta förekommer när vi skall konstruera program.

Abstrakta datatyper

När vi studerar en problemdomän strävar vi efter att identifiera objekt som liknar varandra i något avseende. Vi identifierar sedan de egenskaper hos objekten som har betydelse enligt vår problemdefinition .

Genom att vi kan utelämna sådana attribut och operationer som inte är relevanta kan man säga att vi gör en förenkling av verkligheten. Med C++ har vi möjlighet att göra en implementation som motsvarar vår analys och till att börja med definiera en klass.

Eftersom klassen är en förenklad motsvarighet till något begrepp i verkligheten kan vi betrakta klassen som en abstraktion och använda begreppet "Abstrakt Datatyp".

Vi sammanför attribut och definierar sedan vilka operationer som skall vara möjliga. Dessa definerar klassens gränssnitt och vi skilja detta från implementationen.

Gränssnittet blir då en specifikation för en "Abstract Data Type" (ADT). En egendefinerad abstrakt datatyp (dvs klass) ger vi i första hand egenskaper som gör att den är en bra abstraktion.

När vi använder en abstrakt datatyp är det bara typens gränssnitt som har någon betydelse. Hur implementationen är gjord får inte påverka sättet vi använder den på. Det är inte nödvändigt eller självklart att vi bryr oss om att ge klassen ett gränssnitt som gör att objekten går att använda på samma sätt och i samma sammanhang som de inbyggda datatyperna.

Konkreta datatyper

Ibland finns behov av att ge en datatyp samma eller liknande egenskaper som en inbyggd datatyp. Man vill kanske kunna använda operatorer på ett liknande sätt som med de inbyggda datatyperna och man vill kunna göra deklarationer, funktionsanrop och tilldelningar på samma sätt som
med de inbyggda datatyperna.

En klass som får ett gränssnitt som gör att objekten av denna typ får samma egenskaper som de inbyggda datatyperna kallas en "Konkret Datatyp".

Motiv för en mall för klassdefinitioner

Klassdefinitioner bör innehålla några grundläggande medlemmar. Vi skall nu motivera dessa genom att visa på vad det kan få för konsekvenser att utelämna dem.

Exempel: "En (o)säker array"

Antag att vi har behov av att införa runtimekontroll av indexering i fält. Något som normalt saknas i C++. Vi definierar därför klassen "IntArr" och omdefinierar indexoperatorn för denna typ.

```
class IntArr {
public:
    IntArr(int);
    ~IntArr();
    int& operator[](int);
private:
    int *arr;
    int limit;
};
```

Konstruktor

Vi ger klassen en konstruktor som anropas med ett heltalsargument och som dynamiskt allokerar ett heltalsfält på heapen.

```
IntArr::IntArr(int lim)
:  arr(new int[lim]),limit(lim)
{
}
```

Destruktor

Vi ger också klassen en destruktor som ser till att heltalsfältet deallokeras när vi använder delete på en pekare till ett objekt eller när ett temporärt objekt skall deallokeras.

```
IntArr::~IntArr()
{
     delete [] arr;
}
```

int& IntArr::operator [](int index)

Sedan definierar vi en [ ] -operator för klassen IntArr (Varför måste funktionen vara av typen "int&" ?)

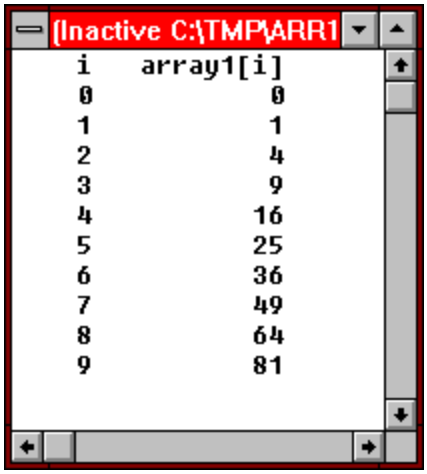
```
int& IntArr::operator [](int index)
{
    if (index < 0 || index >= limit)
    {
      cerr <<"\"IntArr::operator[](int)\"Otillåtet index! ("
           <<index << ")\n";
      exit(1);
    }
    return  arr[index];
}
```

Använda IntArr

```
main()
{
     // Definiera ett arrayobjekt....
     IntArr array1(10);
     // Tilldela värden till elementen....
     for(int i=0;i<10;i++)
     {
          array1[i]=i*i;
     }
     // Skriv ut en tabell
     cout << setw(5) << "i" << setw(12)
          << "array1[i]" << endl;
     for(i=0;i<10;i++)
     {
          cout << setw(5) << i << setw(12)
               << array1[i] <<  endl;
     }
```

```
    return 0;
}
```

Resultat



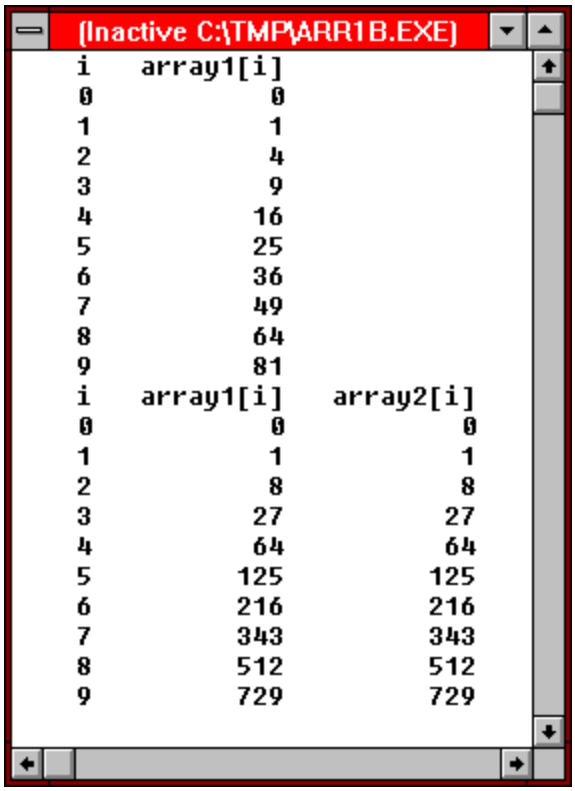
Tilldelning mellan IntArr-objekt

Vad händer om vi använder flera IntArr och börjar göra tilldelningar?

```
main()
{
    // Definiera ett arrayobjekt....
    IntArr array1(10);
    // Definiera ännu ett arrayobjekt...
    IntArr array2(10);

    // Tilldela värden till elementen....
    for(int i=0;i<10;i++)
    {
      array1[i]=i*i;
    }

    // Skriv ut en tabell
    cout << setw(5) << "i";
    cout << setw(12) << "array1[i]"
       << endl;
    for(i=0;i<10;i++)
    {
            cout << setw(5) << i <<
                     setw(12)<< array1[i] <<  endl;
    }

    // Låt det andra vara lika med det första...
    array2 = array1;

    //Manipulera med elementen i den andra arrayen...
    for(i=0;i<10;i++)
    {
      array2[i] = array2[i] * i;
    }
```

```
    // Skriv ut rubriker...
    cout << setw(5) << "i"
       << setw(12) << "array1[i]"
       << setw(12) << "array2[i]"
       << endl;
// Skriv ut en tabell... for(i=0;i<10;i++)

    {
      cout << setw(5)  << i
           << setw(12) << array1[i]
           << setw(12) << array2[i]
           << endl;
    }

    return 0;
}
```

Resultat



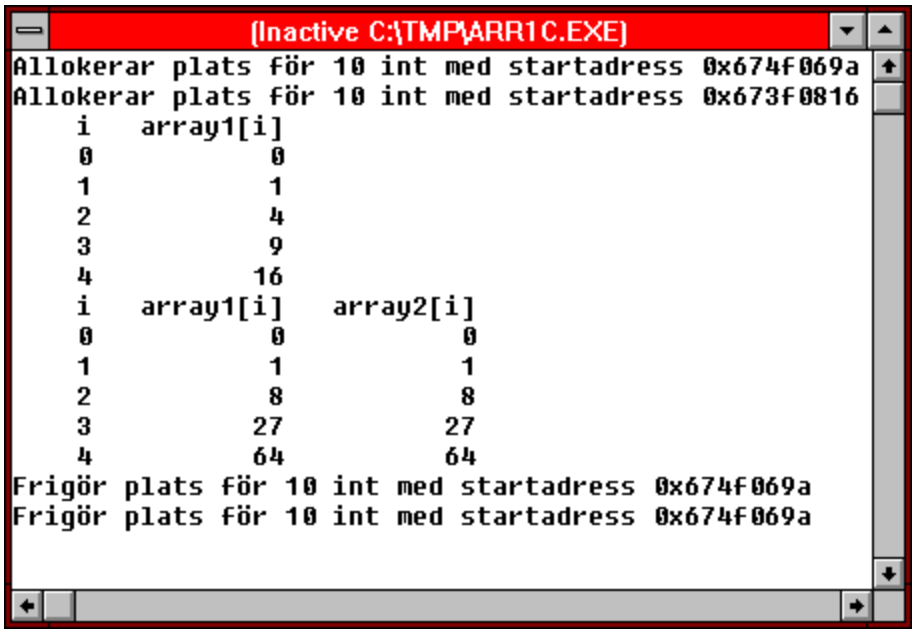
När vi manipulerar med array1 påverkas array2. Varför?

Använd konstruktor och destruktor för avlusning

Vi låter konstruktor och destruktor rapportera vad som händer...

```
IntArr::IntArr(int lim)
: arr(new int[lim]),limit(lim)
{
     cout << "Allokerar plats för " << limit <<
          " int med startadress " << arr << endl;
}
IntArr::~IntArr()
{
     cout << "Frigör plats för " << limit <<
          " int med startadress " << arr << endl;
     delete [] arr;
```

Resultat

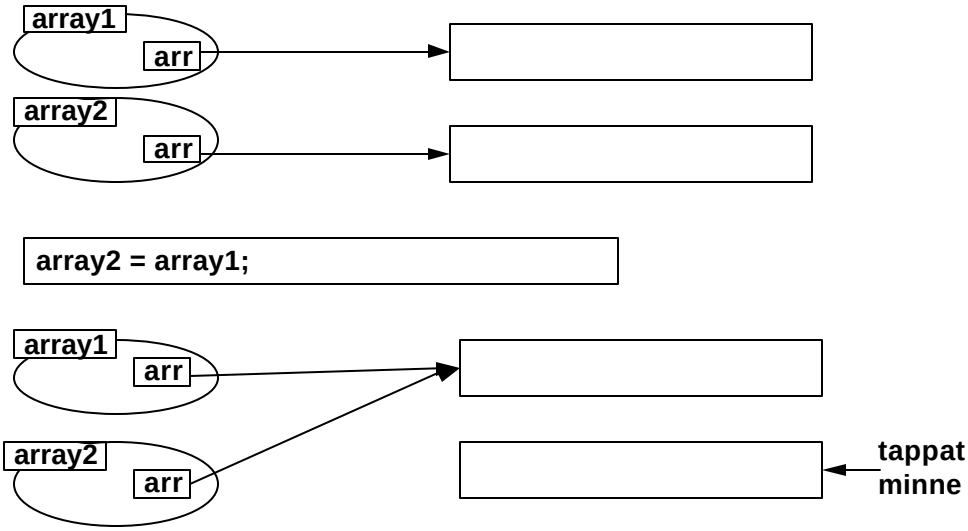


Problem:

- Vi använder "delete" på samma adress två gånger. Detta får oönskade konsekvenser.
- Vi manipulerar med array1 när vi ändrar i array2!
- Vi tappar bort det minne som array1 allokerade

Defaulttilldelning...

Vid tilldelning sker medlemsvis kopiering. Pekaren kopieras men inte det minne som den pekar på:

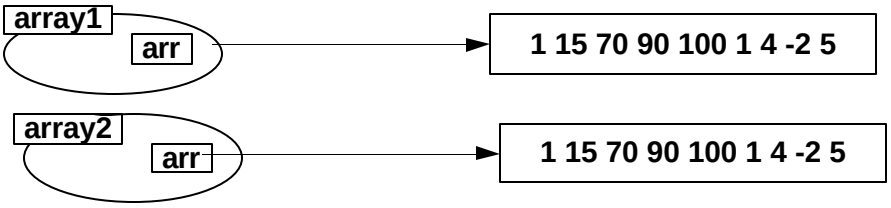

array1
arr
array2
arr
array2 = array1;
array1
arr
array2
arr
tappat
minne

Problem!

Vi måste bestämma oss för om vi vill:

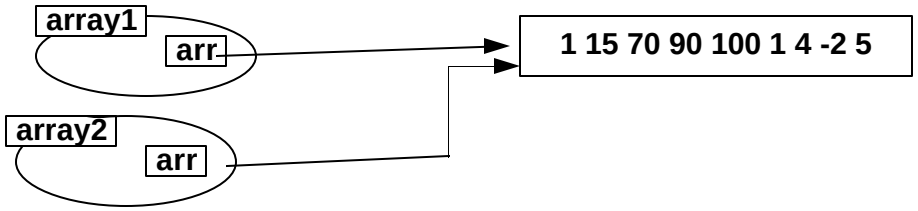
(A) en tilldelning betyder att vi utför en kopiering av det som refereras :

A



(B) en tilldelning skall betyda att två objekt får en referens till samma minne (en resurs) .

B



(A) Tilldelning/initiering innebär kopiering

Vi väljer först att visa hur man implementerar kopiering av objekt som refererar till en resurs (oftast minne).

Vi skall alltså bl.a. undvika att:

> vi använder "delete" på samma adress två gånger samt de oönskade konsekvenser som detta får.
>
> manipulation av ett objekt (array1) innebär att vi även manipulerar med ett annat (array2).
>
> Vi tappar bort det minne som ett objekt allokerar (array1).

(Vi skall senare visa hur man implementerar resursdelning.)

Egendefinierad tilldelningsoperator

Vi löser problemet med hjälp av en egendefinierad tilldelningsoperator.
Tilldelningsoperatorn kan bara överlagras som en medlemsfunktion. Den ärvs inte.

```
class IntArr {
public:
    IntArr(int);
    ~IntArr();
    int& operator[](int);
    IntArr& operator=(const IntArr&);
private:
    int *arr;
    int limit;
};
```

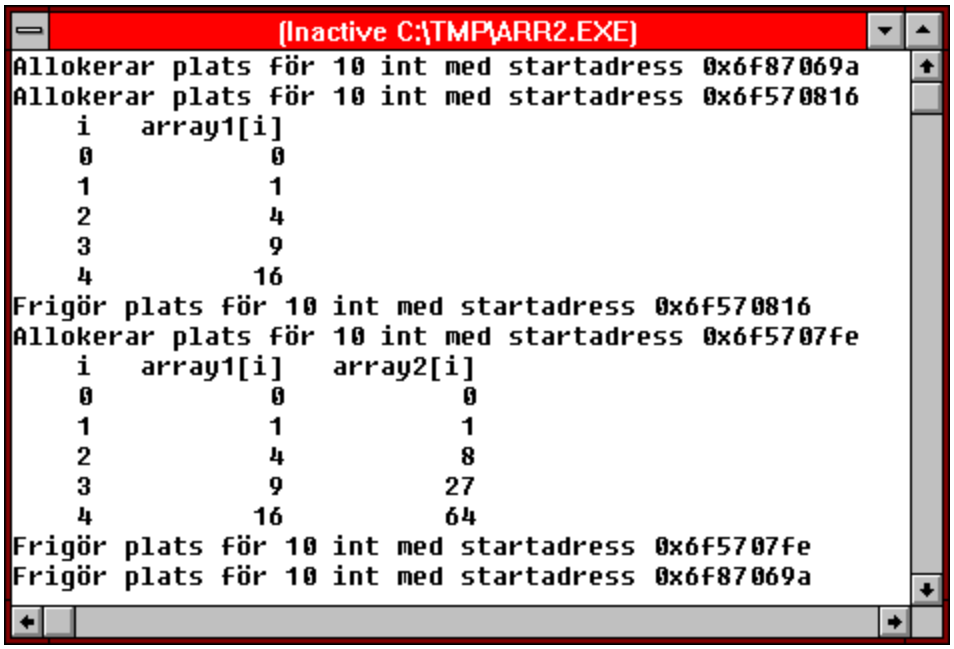
Definition av tilldelningsoperatorn

Vi låter tilldelning betyda att kopian får samma dimension som originalet.
Sedan kopierar vi element för element.....

```
IntArr& IntArr::operator=(IntArr& ia)
{
      if (this != &ia) // vitalt skydd mot självtilldelning!!
        {
            cout << "Frigör plats för " << limit
                 << " int med startadress " << arr << endl;
            delete [] arr; // Frigör det egna fältet...
            limit = ia.limit; // Använd det andra objektets
                              // dimension
            arr = new int[limit]; // Allokera nytt
                      // minne...
            for(int i=0;i<limit;i++) // Kopiera in elementen
                  (*this)[i]= ia[i];
            cout << "Allokerar plats för " << limit
                 << " int med startadress " << arr << endl;
        }
        return *this;        // Returnera mig själv...
}
```

Resultat

Nu har vi löst problemet med tilldelning....



Initiering

Antag att vi vill kunna initiera ett nytt objekt med ett annat objekt. Detta vore praktiskt och vi är vana vid att kunna göra detta med andra typer. Man skulle kunna anta att detta går bra eftersom vi har infört en tilldelningsoperator.

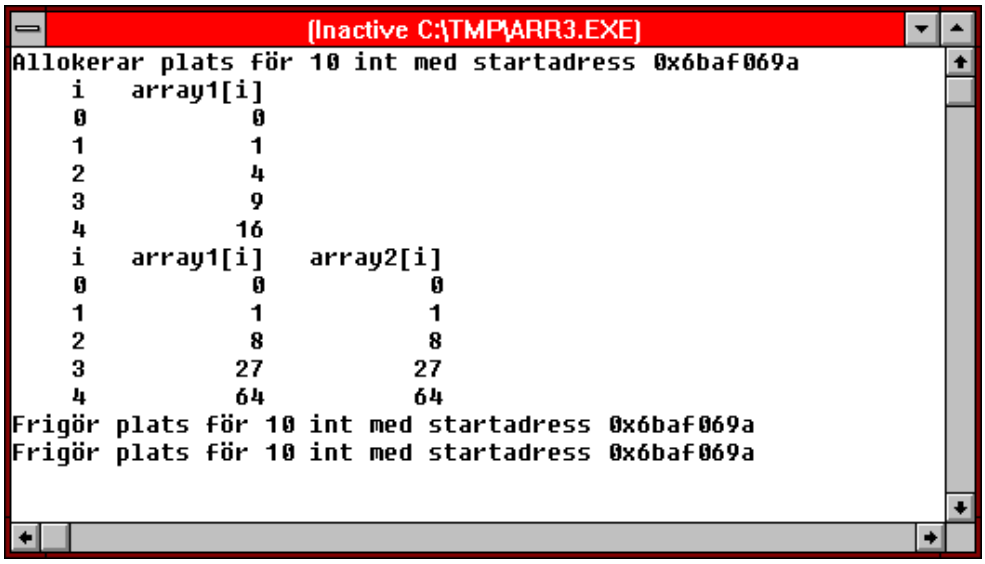
```
main()
{
    // Definiera ett arrayobjekt....
    IntArr array1(10);

    // Tilldela v„rden till elementen....
    for(int i=0;i<10;i++)
    {
      array1[i]=i*i;
    }

    cout.fill(' ');
    // Skriv ut en tabell
    cout << setw(5) << "i";
    cout << setw(12) << "array1[i]"
       << endl;
    for(i=0;i<5;i++)
    {
            cout << setw(5) << i
               << setw(12) << array1[i] <<  endl;
    }

      // Definiera ännu ett arrayobjekt och
      // initiera detta med det första..
    IntArr array2 = array1;
```

```
    // Manipulera med elementen i den andra arrayen...
    for(i=0;i<10;i++)
    {
      array2[i] = array2[i] * i;
    }

    cout.fill(' ');
    // Skriv ut rubriker...
    cout << setw(5) << "i"
       << setw(12) << "array1[i]"
       << setw(12) << "array2[i]"
       << endl;

    // Skriv ut en tabell...
    for(i=0;i<5;i++)
    {
      cout << setw(5)  << i
           << setw(12) << array1[i]
           << setw(12) << array2[i]
           << endl;
    }

    return 0;
}
```

Resultat

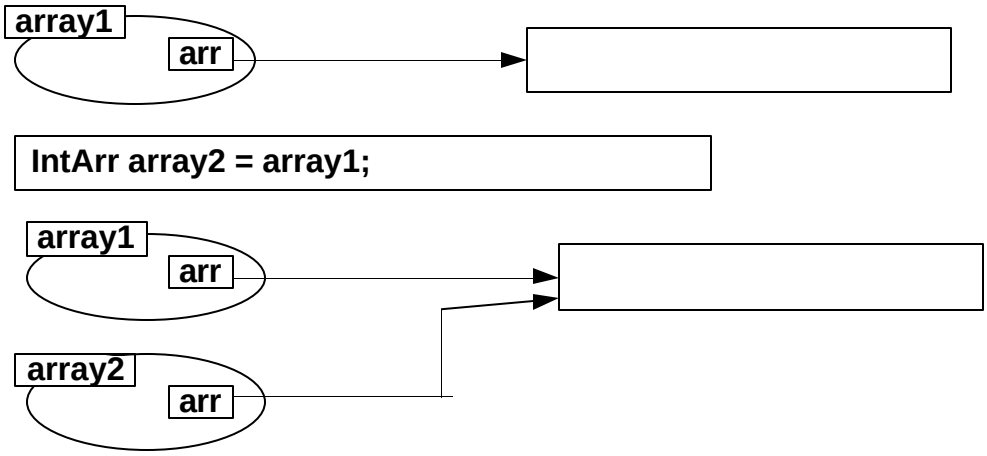
Det visar sig dock att vi får samma problem som förut dvs att pekarens värde kopieras.

Slutsats: tilldelningsoperatorn används inte vid initiering.



Defaultinitiering....

Vid initiering sker medlemsvis kopiering.Vi allokerar inte något eget minne för "arr" i array2 och vi kommer att använda delete två gånger på samma adress (i destruktorn).



Copy-konstruktor

Vid initieringar genererar kompilatorn en default konstruktor som utför medlemsvis kopiering. Som tur är kan vi definiera en egen "Copy-konstruktor" som gör allt på rätt sätt. Lägg märke till hur den är deklarerad:

```
class IntArr {
public:
    IntArr(int);   //
    IntArr(const IntArr& ia);//"Copy"-konstruktor
    ~IntArr();
    int& operator[](int);
    IntArr& operator=(const IntArr&);
private:
    int *arr;
    int limit;
};
```

Definition av Copy-konstruktorn

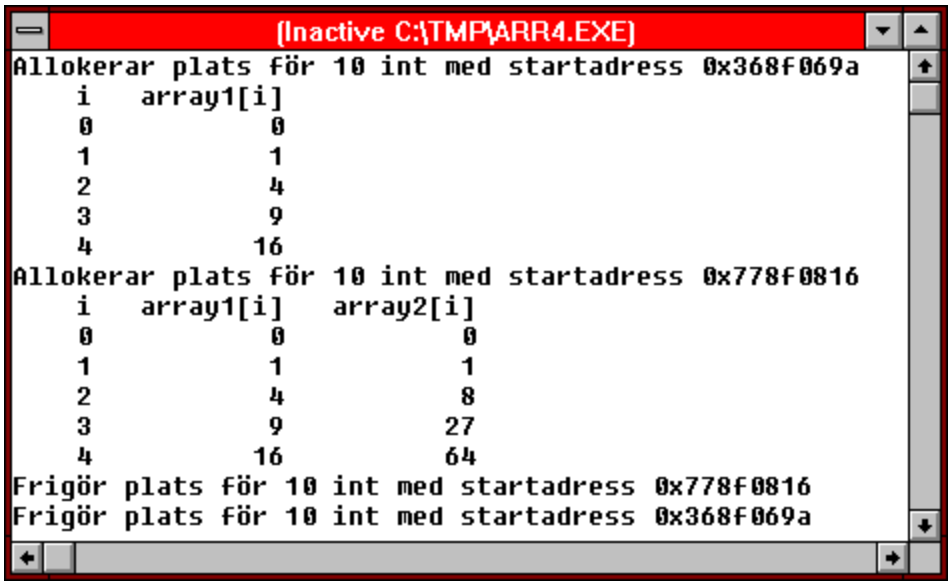
I definitionen av klassens Copy-konstruktor använder vi samma dimension som orginalet men allokerar ett nytt heltalsfält. Sedan kopierar vi in elementen med hjälp av tilldelningsoperatorn:

```
IntArr::IntArr(const IntArr& ia)
: arr(new int[ia.limit]),limit(ia.limit) // Allokera och initiera...
{
      cout << "Allokerar plats för " << limit
           << " int med startadress " << arr << endl;
      for(int i=0;i<limit;i++)    // Kopiera in elementen
```

```
            (*this)[i]= ia[i];  // ett och ett......
}
```

Resultat

När vi prövar vårt program visar det sig att vi har löst problemet. Initiering innebär nu kopiering!



Konstruktor och funktionsanrop

Antag att vi skriver en funktion som skriver ut värden från en "IntArr". Vad har det för betydelse om vi definierar den som funk1 (värdeanrop) eller funk2 (referensanrop) nedan och vad krävs för att vi skall kunna använda ett konstant objekt av typ IntArr på det sätt som visas nedan?

```
void funk1(const IntArr array, int n)
{
      cout.fill(' ');
      // Skriv ut en tabell
      cout << setw(5) << "i" << setw(12) << "array1[i]"
           << endl;
      for(int i=0;i<n;i++)
      {
            cout << setw(5) << i <<
                    setw(12) << array[i]  <<  endl;
      }
}

void funk2(const IntArr& array, int n)
{
      cout.fill(' ');
      // Skriv ut en tabell
      cout << setw(5) << "i" <<
```

```
                setw(12) << "array[i]" << endl;
        for(int i=0;i<n;i++)
        {
              cout << setw(5) << i <<
                      setw(12) << array[i] <<  endl;
        }
}
```

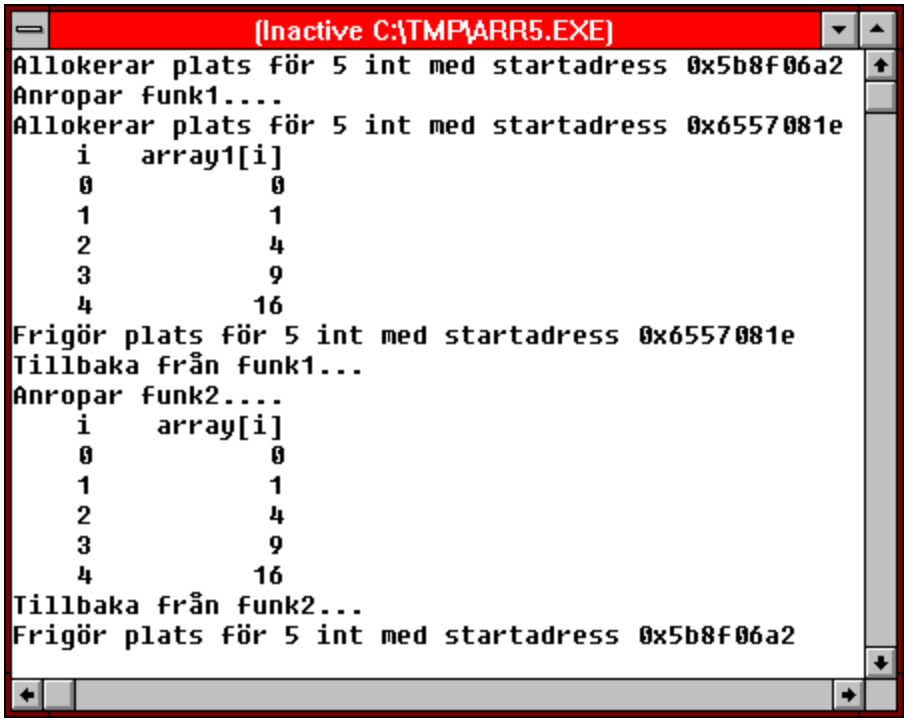
Medlemsfunktioner för konstanta objekt

För att vi skall kunna använda konstanta IntArr-objekt måste vi införa ytterligare en överlagrad []-operator.

```
class IntArr {
public:
    IntArr(int);   //
    IntArr(const IntArr& ia); // "Copy"-konstruktor
    ~IntArr();
    int& operator[](int);
    int  operator[](int) const;
    IntArr& operator=(const IntArr&);
private:
    int *arr;
    int limit;
};
```

Resultat

Copy-konstruktorn används när vi anropar funktioner med värdeanrop men inte när vi använder referensanrop (det sker ingen kopiering då).



Slutsats

En value-class bör som regel redan från början förses med följande:

1. En default konstruktor
2. En Copy-konstruktor
3. En tilldelningsoperator
4. En destruktor.

Det kan hända att någon av ovanstående är uppenbart onödig eller ger uppenbara nackdelar men programmeraren bör vara medveten om vilka konsekvenser det kan innebära att någon av dem saknas.

STANDARDMALL

```
// En standardmall för värdeklasser
class A {
public:
     // Standardkonstruktorer
     A(void); // Default konstruktor
     A(A&);   // Copy - konstruktor
     // Tilldelningsoperator
     A& operator=(const A&);
     // Destruktor
     ~A();
      /* Här definieras ”övriga metoder mm. */
     // ....
protected:
     // ...
private:
     // ....
};
```

När skall standardmallen användas?

Konkreta datatyper skall följa standardmallen för att på ett fullgott sätt integreras med andra standardtyper och för att kunna användas på samma sätt som dessa (funktionsanrop, initieringar, tilldelningar mm).

Abstrakta datatyper behöver inte alltid följa mallen men bör göra det om:

- du vill göra tilldelningar
- du vill skicka objekt genom värdeanrop
- objekten har en pekare till något
- om destruktorn använder delete på en datamedlem

Du skall använda standardmallen för alla icketriviala klasser

(B) Tilldelning/initiering innebär delning

Vi ska nu visa en modell för hur man i stället för att kopiera i samband med tilldelning och initiering medvetet kan välja att låta objekten dela en resurs (t.ex. referera till samma minnesutrymme). Vi skall fortfarande bl.a. undvika att:

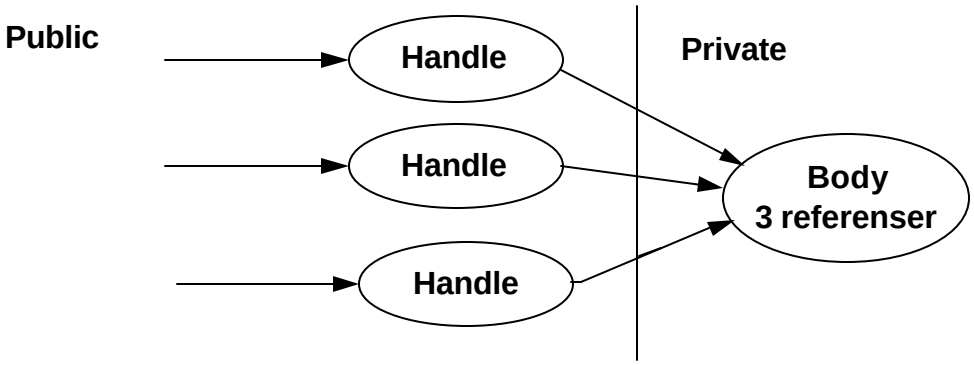
vi använder "delete" på samma adress två gånger samt de oönskade konsekvenser som detta får.

vi tappar bort det minne som ett objekt allokerar (array1).

Däremot kommer manipulation av ett objekt att innebära att detta påverkar andra objekt som refererar till samma resurs. Detta kan leda till att vi väljer att betrakta resurser som konstanta.

Referensräknare:exempel

Vi kommer att använda en metod som brukar gå under beteckningen Handle-Body-idiomet". Vi kommer att använda två klasser. Den ena kommer att vara gränsnitt (handle) mot den andra som motsvarar själva resursen (body).



Bodyklass

Vår bodyklass har bara inkapslade medlemmar och har **ResursHandle** som vänklass. Vi kan alltså bara komma åt en resurs via en handle.

```
// Detta är en "resurs" som skall administreras..
class ResursBody {
     friend ResursHandle;
private:
     ResursBody();
     ~ResursBody();
```

```
     long number;            // referensräknare
     static long totalnumber;    // För kontroll
};
```

Så här har vi definierat medlemsfunktionerna....

```
long ResursBody::totalnumber = 0;
ResursBody::ResursBody()
: number(1)
{
     totalnumber++;
}
ResursBody::~ResursBody()
{
     totalnumber--;
}
```

Handleklass

Vår handleklass har medlemmar för initiering och tilldelning samt en pekare till en resurs.

```
class ResursHandle {
public:
     ResursHandle();
     ResursHandle(const ResursHandle& rh);
     ResursHandle& operator=(const ResursHandle& rh);
     ~ResursHandle();
     void dumpthis() const;
     static void dumptotal();
private:
     ResursBody    *body;
     static long     totalnumber;
};
```

Nedan visas hur medlemsfunktionerna är definierade...

```
long ResursHandle::totalnumber = 0;
ResursHandle::ResursHandle()
{
      body = new ResursBody;
      totalnumber++;
}
ResursHandle::ResursHandle(const ResursHandle& rh)
{
      body = rh.body;
      rh.body->number++;
      totalnumber++;
}
ResursHandle& ResursHandle::operator=(const ResursHandle& rh)
{
      if (this!=&rh)
      {
            rh.body->number++;
            if (--body->number <= 0)
                  delete body;
            body = rh.body;
      }
      return *this;
}
ResursHandle::~ResursHandle()
{
      if (--body->number<=0)
            delete body;
      totalnumber--;
}
void ResursHandle::dumpthis() const
{
      cout <<"Detta är ett handtag till en resurs på adress "
         << body <<" med "<<body->number <<" referens(er)\n";
}
void ResursHandle::dumptotal ()
{
      cout << "Det finns totalt " << ResursBody::totalnumber
          << " resurs(er) och "  << ResursHandle::totalnumber        <<
" handtag allokerade\n";
}
```

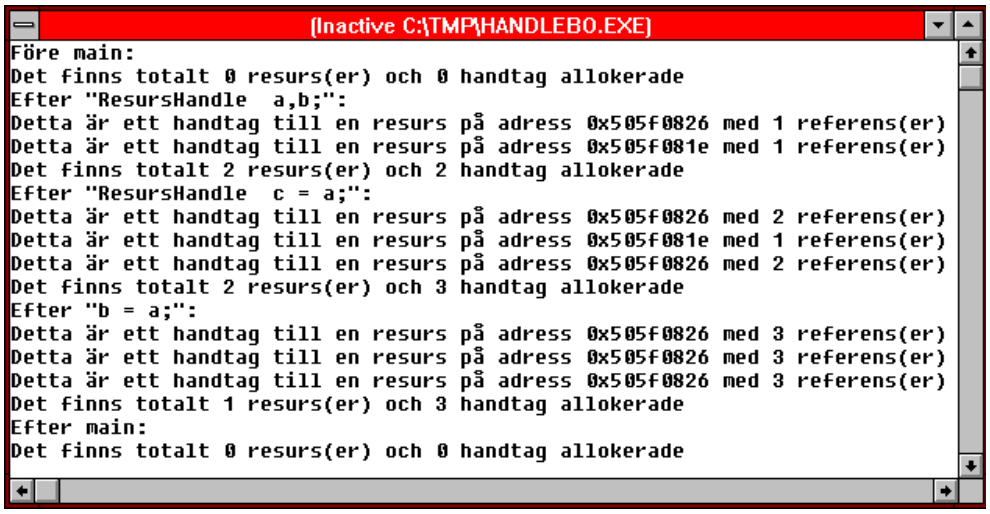
Dessutom använder vi en klass för kontroll av antalet objekt före och efter main: Vi skapar ett globalt objekt av typ Kontroll. Klassens konstruktor anropas före main och dess destruktor efter main.

```
class Kontroll {
public:
     Kontroll() {
          cout << "Före main:\n";
          ResursHandle::dumptotal();
     }
     ~Kontroll() {
          cout << "Efter main:\n";
          ResursHandle::dumptotal();
     }
} dummy;
```

I main prövar vi att skapa objekt samt att göra tilldelningar och initieringar....

```
main()
{
     ResursHandle  a,b;      // Två objekt
     cout << "Efter \"ResursHandle  a,b;\":\n";
     a.dumpthis(); b.dumpthis();
     ResursHandle::dumptotal();

     ResursHandle  c = a;    // Initiering...
     cout << "Efter \"ResursHandle  c = a;\":\n";
     a.dumpthis();b.dumpthis();c.dumpthis();
     ResursHandle::dumptotal();

     b = a;                                    //
Tilldelning
     cout << "Efter \"b = a;\":\n";
     a.dumpthis(); b.dumpthis();c.dumpthis();
     ResursHandle::dumptotal();

     return 0;
}
```

Resultat


(Inactive C:\TMP\HANDLEBO.EXE)
Före main:
Det finns totalt 0 resurs(er) och 0 handtag allokerade
Efter "ResursHandle a,b;":
Detta är ett handtag till en resurs på adress 0x505f0826 med 1 referens(er)
Detta är ett handtag till en resurs på adress 0x505f081e med 1 referens(er)
Det finns totalt 2 resurs(er) och 2 handtag allokerade
Efter "ResursHandle c = a;":
Detta är ett handtag till en resurs på adress 0x505f0826 med 2 referens(er)
Detta är ett handtag till en resurs på adress 0x505f081e med 1 referens(er)
Detta är ett handtag till en resurs på adress 0x505f0826 med 2 referens(er)
Det finns totalt 2 resurs(er) och 3 handtag allokerade
Efter "b = a;":
Detta är ett handtag till en resurs på adress 0x505f0826 med 3 referens(er)
Detta är ett handtag till en resurs på adress 0x505f0826 med 3 referens(er)
Detta är ett handtag till en resurs på adress 0x505f0826 med 3 referens(er)
Det finns totalt 1 resurs(er) och 3 handtag allokerade
Efter main:
Det finns totalt 0 resurs(er) och 0 handtag allokerade

Övningar

Övning 1

Antag att vi kommer att göra ett lagersystem där lagerartiklarna har ett idnr, ett namn, en beskrivning, ett pris samt antal artiklar på lagret. Implementera en klass som kan användas för att hantera **LagerObjekt.** Namn och beskrivning utgörs av pekare till godtyckligt stora teckenfält på heapen..

Ge klassen defaultkonstruktor, copykonstruktor, tilldelningsoperator och destruktor samt övriga metoder, konstruktorer och attribut som du anser behövs.

Försök sedan att verifiera att objekt av din typ går att använda vid tilldelning och initiering på ett riskfritt sätt (inklusive funktionsanrop).

Övning 2

I ett program förekommer ofta strängar som är och skall vara konstanta. Om dessa används vid tilldelning och initiering av andra konstanta strängar är det onödigt att utföra kopiering.

Inför typen **ConstStreng** som en handleklass till klassen **Streng** som skall vara en "standard strängklass". Inför referensräkning och se till att kopiering av en **ConstStreng** inte innebär kopiering av en **Streng** utan att antalet referenser räknas upp. På samma sätt skall ett **Streng**-objekt frigöras endast om antalet referenser är noll.

Ge klassen **ConstStreng** defaultkonstruktor, copykonstruktor, tilldelningsoperator (?) och destruktor samt övriga metoder, konstruktorer och attribut som du anser behövs.

Försök sedan att verifiera att objekt av din typ går att använda vid tilldelning och initieringar på ett riskfritt sätt (inklusive funktionsanrop). Pröva också att anropa strlen, strcpy och liknande funktioner med objekt av typen **ConstStreng** som parametrar.

Övning 3

Implementera en klass för en länkad lista för objekt av t.ex. typen **LagerObjekt** som du gjorde i den första övningen. Klassen kan t.ex. heta **LagerLista**.

Ge klassen **LagerLista** defaultkonstruktor, copykonstruktor, tilldelningsoperator och destruktor samt övriga metoder, konstruktorer och attribut som du anser behövs. Försök sedan att verifiera att objekt av din typ går att använda vid tilldelning och initieringar på ett riskfritt sätt (inklusive funktionsanrop).

Övning 4

Antag att det i ett program förekommer konstanta länkade listor av typen **LagerLista** som du gjorde i föregående uppgift.

Inför klassen **ConstLagerLista** som skall vara en handleklass till **LagerLista.** Om dessa används vid tilldelning och initiering av andra listor är det onödigt att utföra kopiering.

Inför referensräkning och se till att kopiering av en **ConstLagerLista** inte innebär kopiering av en utan att antalet referenser räknas upp. På samma sätt skall ett **LagerLista**-objekt frigöras endast om antalet referenser är noll.

Ge klassen **ConstLagerLista** defaultkonstruktor, copykonstruktor, tilldelningsoperator och destruktor samt övriga metoder, konstruktorer och attribut som du anser behövs.

Försök sedan att verifiera att objekt av din typ går att använda vid tilldelning och initieringar på ett riskfritt sätt (inklusive funktionsanrop).