Innehållsförteckning

1 Pekare

I detta avsnitt ska vi diskutera en av C++ mest kraftfulla konstruktioner, pekare. Vi kommer längre fram att visa hur man med pekare kan skapa dynamisk allokering av data, se hur pekare och arrayer (arrayer) är släkt med varandra och utnyttja kraften i pekare då vi skapar länkade listor.

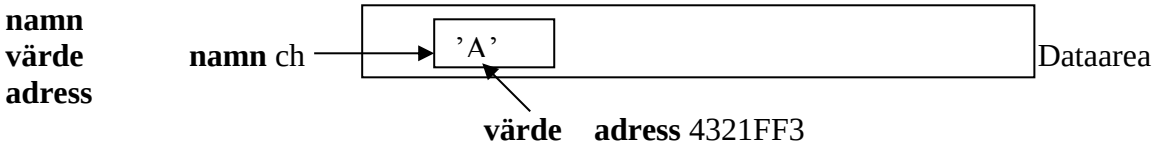
1.1 Variabler i minnet

Först ska vi repetera och fördjupa hur vi hanterar variabler av de enkla datatyperna i C++.

När vi deklarerar variabler knyter vi en del av minnet till variabeln. Denna del av minnet kallas dataarea eller stack. Minnesutrymmets storlek som variabeln är knuten till beror på datatypen. I detta fall får ch en byte. För att hantera variabeln har operativsystemet en lista som har tre fält:

variabelnamn	**värde**	**minnesadress**
ch	’A’	4321AFF3 (hex)

Detta behöver vi inte bekymra oss om, kompilatorn sköter detta åt oss. För att förstå pekare kan det dock underlätta om tänker på en variabel som om den har tre fält:



Ytterligare exempel:
int tal =123;
char ch = ’A’;
double radie = 1.2345;
float bredd = 35.6;

Variabelnamn	**värde**	**adress**		
i	123	00AE1000	int	
		00AE1001		
		00AE1002		
		00AE1003	4 byte	
ch	‘A’	00AE1004	1 byte	char
radie	1.2345	00AE1005	double	
		00AE1006		
		00AE1007		
		00AE1008		
		00AE1009		
		00AE100A		
		00AE100B		
		00AE100C	8 byte	
bredd	35.6	00AE100D	float	
		00AE100E		
		00AE100F		
		00AE1010	4 byte	

32 – bitars-adresser gäller för Windows XP. Äldre OS och programvaror kan ha 16 –bitars eller 20 – bitars-adresser.

1.2 Deklarera pekare

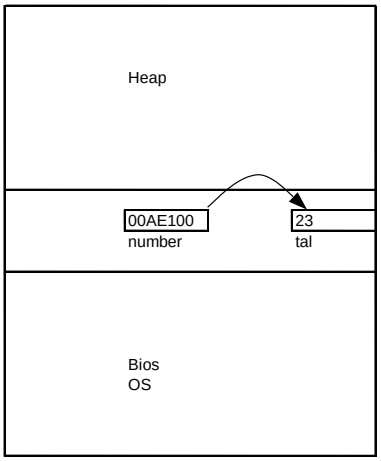
En pekare är en variabel som innehåller en adress till data.

```
int *number;
float *radius;
```

number är en pekare till en **int**.
radius är en pekare till en **float**.
Båda saknar värde i detta läge dvs dom har ingen adress (ingenstans att peka).

Ett sätt att ge adresser till dessa pekare är att använda befintliga variablers adresser:

```
int tal = 23;
number = &tal;
float r = 12.3;
radius = &r;
```



där & = ”ta adressen av”
& kallas **adressoperatorn**

Detta betyder att ’number’ pekar på det minnesutrymme där ’tal’ ligger . Pss för ‘radius’ och ‘r’ .
För att hantera det värde som number pekar på skriver man:

```
cout << *number;
```

Termer:

Pekaroperatorn	*	int *number;	
Innehållsoperatorn	*	cout << *number;	(dereferensoperatorn)
Adressoperatorn	&	number = &i;	(referensoperatorn”, ta adressen av”)

Programexempel:

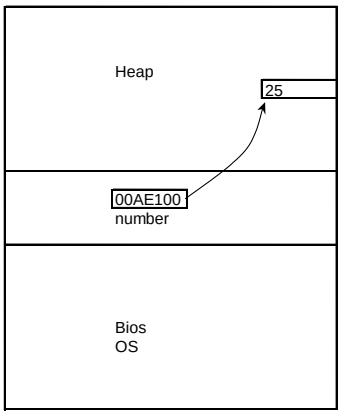
```
// point_010 Version 10
// Deklarera en pekare
// Ta adressen av
// Per Ekeroot 2014-01-13
//---------------------------------------------------------------------------
#include <iostream>   // cout
using namespace std;
//---------------------------------------------------------------------------
int main()
{
  // Deklarera och initiera en heltalsvariabel
  int num = 12;

  // Deklarera och initiera en pekarvariabel
  int *pnum = NULL;

  // Tilldela en adress till pekarvariabeln
  pnum = &num;

  cout << " num   = " << num << endl;   // Skriv heltalsvariabeln
  cout << " pnum  = " << pnum << endl;  // Skriv pekarens adress
  cout << " *pnum = " << *pnum << endl; // Skriv värdet som ligger i minnescellen
                                        // på vilken pekarvariabeln pekar

  return 0;
}
```

Kommentarer:

- int **pnum* =NULL allokerar plats för en adress, vilken tilldelas NULL (=0). Detta innebär att den pekar på ingenting.
- *pnum* = &*num*; tilldelar adressen som num har till pnum
- *pnum* innehåller adressen till *num*
- **pnum* ger värdet som ligger i minnescellen på vilket *pnum* pekar

1.3 Dynamisk allokering av plats för data

Ett annat sätt att ge en pekarvariabel en adress är att skapa (allokera) en plats på heapen (det fria lagringsutrymmet). Detta görs dynamiskt medan programmet körs.




```
int *number = new int;
*number = 25;
```

På första raden allokeras plats för ett heltal (int) på heapen.
Minnescellens adress läggs i *number*. På andra raden läggs heltalet 25 i den minnescell som *number* pekar på.

För att frigöra det allokerade utrymmet används `delete`

```
delete number;
```

Det är mycket viktigt at man kommer ihåg att frigöra det minne man har allokerat. Om man inte frigör minne så blir detta minne låst så att inga andra processer kan använda det. Man brukar säga att man får en minneslucka!

Ett programexempel:

```
// point_020 Version 10
// Deklarera pekare dynamiskt
// Per Ekeroot 2014-01-13
//-----------------------------------------------------------------------------
#include <iostream>   // cout
using namespace std;

//-----------------------------------------------------------------------------
int main()
{
  int *number = new int;   // Deklarera en pekare som pekar på en int
  *number = 25;            // Initiera pekaren

  // Skriv pekarens värde och dess adress
  cout << "The numbers value             = " << *number << endl;
  cout << "The addressen to the number   = " << number << endl << endl;

  double *number2 = new double;    // Deklarera en pekare som pekar på en double
  *number2 = 12.34;                // Initiera pekaren

  // Skriv pekarens värde och dess adress
  cout << "The value of number2          = " << *number2 << endl;
  cout << "The address of number 2       = " << number2 << endl << endl;

  // Skriv utrymmesbehov för pekare resp det pekaren pekar på
  cout << "The size of number            = " << sizeof *number << endl;
  cout << "The size of numbers address   = " << sizeof number << endl << endl;

  cout << "The size of number2           = " << sizeof *number2 << endl;
  cout << "The size of number2's address = " << sizeof number2 << endl << endl;
```

```
  // Frigör platsen som number och number2 upptar på heapen
  delete number;
  delete number2;

  return 0;
}
```

Kommentarer:

- Pekare allokeras och initieras
- Pekarens värde och adress skrivs
- Pekarnas platsbehov skrivs
  - sizeof number ger adressens platsbehov
  - sizeof *number ger platsbehovet för det number pekar på, en int i detta fall.

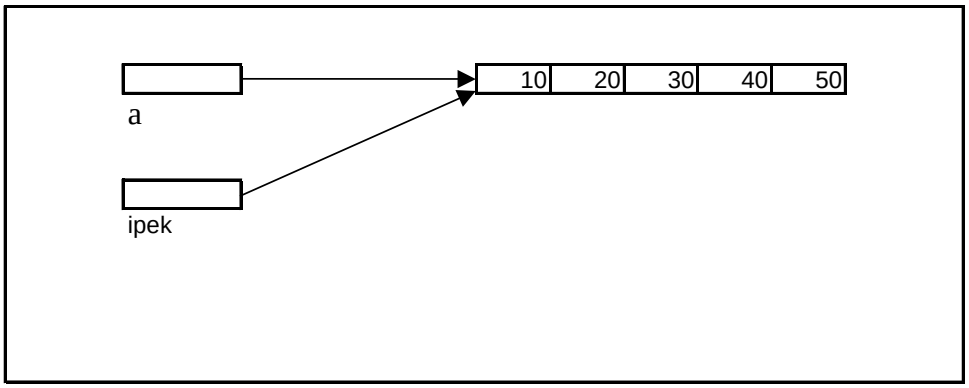
Vi återkommer längre fram till allokering av plats för arrayer mm.

1.4 Pekare och arrayer

Det finns ett nära släktskap mellan pekare och arrayer. Det förstår vi av att en arrays namn = adressen till arrayens första element. Detta åskådliggörs i följande exempel där en array och en pekare deklareras och "kopplas" ihop.

```
int a[] = {10, 20, 30, 40, 50};
int *ipek;
ipek = a;
```

I första satsen deklareras och initieras en array. I den andra satsen deklareras en pekare till heltal. Sedan låter vi pekaren peka på arrayens första element genom att ipek (=adress) tilldelas arrayens namn (= arrayens adress).



För att komma åt elementen i arrayen kan man använda indexnotation eller pekarnotation.

1. Indexnotation: `cout << a[2];` skriver talet 30
2. Pekarnotation: `cout << *(ipek + 2)` skriver talet 30

Variant 1 är vi vana vid, men hur fungerar alternativ 2?

- Pekararitmetik utförs alltid relativt storleken på den typ som pekaren är deklarerad att peka på!
- Dvs addition/subtraktion etc på en pekarvariabel (adress) görs i enheten **sizeof** (datatypen som pekaren pekar på).
- Ex **ipek** pekar på en int vilket medför att **ipek + 2** pekar 2*4 bytes längre fram i minnet

Man kan dessutom använda dessa notationssätt **omväxlande**! Se nedan!

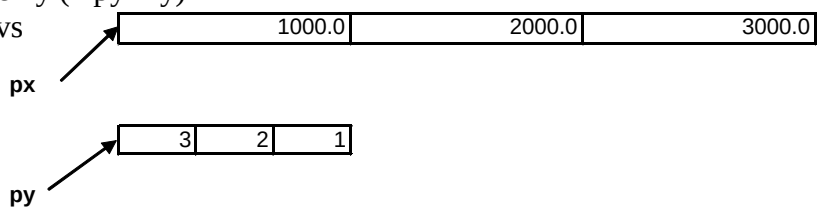
```
for(int i=0; i<5; i++)
  cout << x[i] << endl;

for(int i=0; i<5; i++)
  cout << ip[i] << endl;

for(int i=0; i<5; i++)
  cout << *(x + i) << endl;

for(int i=0; i<5; i++)
  cout << *(ip+i) << endl;
```

Programexempel:

```
// point_030 Version 10
// Pekare och arrayer (vektorer)
// Per Ekeroot 2014-01-13
//-----------------------------------------------------------------------------
#include <iostream>   // cout
using namespace std;

//-----------------------------------------------------------------------------
int main()
{
  // Deklarera och initiera två arrayer
  double x[3] = {1000.0, 2000.0,3000.0};
  int    y[3] = {3, 2, 1};

  // Två sätt att få adressen till en array
  double *px = x;
  int    *py = &y[0];

  cout << "px = " << px << " , *px = " << *px << endl;
  px = px + 1;
  cout << "Add one (1) to the pointer px (the address): " << endl;
  cout << "px = " << px << " , *px = " << *px << endl << endl;

  cout << "py = " << py << " , *py = " << *py << endl;
  py++;  //py = py + 1;

  cout << "Add one (1) to the pointer py (the address): " << endl;
  cout << "py = " << py << " , *py = " << *py << endl << endl;

  cout << "Get two elements using index: " << endl;
  cout << " x[0]= " << x[0]  << "  x[1]= " << x[1] << endl ;

  cout << "Get two elements using pointer: " << endl;
  cout << " *x= " << *x << "  *(x+1)= " << *(x+1) << endl << endl;

  cout << "Size of the array x    = " << sizeof x << endl;
  cout << "Size of the pointer px = " << sizeof px << endl << endl;

  return 0;
}
```

Kommentarer:

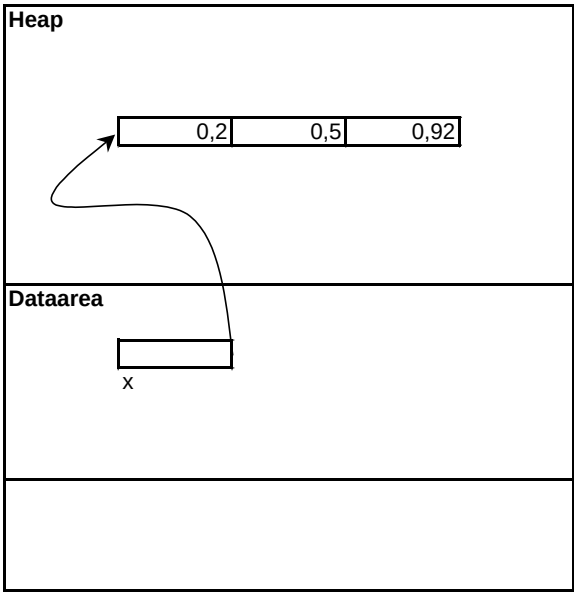
- *px pekar på första elementet i arrayen x (*px = x)
- *py pekar på första elementet i arrayen y (*py = y)
- px = px +1 flyttar pekaren 8 bytes, dvs till nästa element
- py++ flyttar pekaren 4 bytes, dvs till nästa element



1.5 Dynamisk allokering av en array (array)

Vi ska nu visa ett exempel som dynamiskt allokerar plats för en array på heapen. Vi använder sedan indexnotation för att hantera arrayens element och pekarnotation för att stega oss fram i arrayen.

```
double *x = new double [3];
```



Man kan också låta användaren ange arraynes storlek medan programmet körs:

```
  size_t size;
  cout << "Input size of array: ";
  cin >> size;
  int *v = new int[size];          // Allokera plats för en array med size element
```

```
// point_040 Version 10
// Allokera plats dynamiskt för en vektor
// Per Ekeroot 2014-01-13
//-----------------------------------------------------------------------------
#include <iostream>   // cout
#include <iomanip>    // setw()
using namespace std;
//-----------------------------------------------------------------------------
int main()
{
//-----------------------------------------------------------------------------
// Allokera plats för en array som innehåller 3 element med en double i varje.
// Adressen till första elementet läggs i pekaren x.
//-----------------------------------------------------------------------------
  double *x = new double[3];
  x[0] = 0.2;                              // Använd index
  x[1] = 0.5;
  x[2] = 0.92;
  cout << "x[0]= " << x[0] << " x[1]= " << x[1]  << "  x[2]= " << x[2] << endl;
  x = x + 1 ;                               // Stega upp pekaren ett steg
                                            // x pekar nu på "0.5"-elementet
  cout << "x[0]= " << x[0] << " x[1]= " << x[1] << endl<< endl << endl;

  // Frigör platsen där arrayen ligger
  x = x - 1; // För att x ska peka till första elementet i vektorn !!
  delete []x;                               // Frigör allokerad plats
```

```
//--------------------------------------------------------------------------
// Skapa en array efter att användaren angivit arrayens storlek.
//--------------------------------------------------------------------------
  size_t size;
  cout << "Input size of array: ";
  cin >> size;
  int *v = new int[size];         // Allokera plats för en array med size element

  for(size_t i=0; i<size; i++)    // Lägg in värden i arrayen
    v[i] = i;

  for(int i=0; i<size; i++)       // Skriv ut arrayen
  {
     cout << setw(5) << v[i];
     if((i+1)%10==0)
       cout << endl;
  }
          cout << endl << endl;

  delete []v;                     // Frigör allokerad plats
  return 0;
}
```

Kommentarer:

- `double *x = new double[3];` allokerar plats på heapen för en array med 3 element. I varje element ligger en double
- Använd indexnotation vid hantering av arrayen
- `delete []x` och `delete []v` frigör den plats som allokerats för respektive array. Observera skrivsättet med []!
- Man bör använda `size_t` som datatyp för arrayers index. `size_t` är en maskinspecifik `unsigned int` som är tillräckligt stor för att ange storleken för ett objekt (array) i datorns minne.

1.6 En array med pekare

I detta avsnitt visas hur man kan effektivisera sortering av stora datamängder. Vi ska skapa en array med många element och i varje element ska vi lägga en post med ganska mycket data.

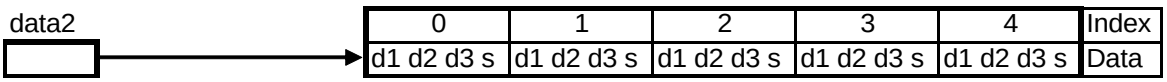
Vi skapar en post:

```
struct Data
{
  double d1;
  double d2;
  double d3;
  string s;
};
```

Jag har valt att låta posten innehålla tre flyttal och en sträng. Speciellt strängen tar mycket processorkraft om den måste flyttas i minnet. Sedan skapar vi en array med posten (Data) i varje element:

```
Data *data2 = new Data[SIZE];
```

Med denna sats gör vi plats för SIZE stycken poster av datatypen Data. Dessa poster ligger i en följd i minnet så här:



Arrayens namn data2 innehåller adressen till första posten. Om denna array ska sorteras så måste posterna flyttas mellan elementen, vilket är resurskrävande. Ett sätt att effektivisera sorteringen är deklarera arrayen så här:

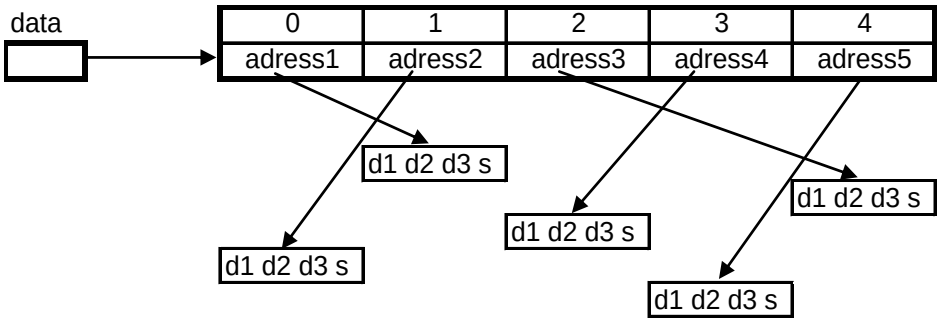
```
  Data *data[SIZE] = {NULL};
```

Detta är en array som innehåller pekare (adresser) i elementen. För att lägga in en adress i ett element och samtidigt allokera plats för en post skriver man :

```
    data[i] = new Data;
```

där i är elementnumret.

En bild av denna array ser ut ungefär så här:



Vid sortering av denna array så flyttas inte posterna, utan det är enbart pekarna som flyttas.
Att flytta pekaren kräver betydligt mindre resurser än att flytta posterna!
Lägg in data på den plats som allokerats:

```
    data[i]->d1 = random(10000)/100.0;
    data[i]->d2 = random(10000)/100.0;
    data[i]->d3 = random(10000)/100.0;
    data[i]-> s = "Kalle Svensson i skogen";
```

i är fortfarande index till elmentet.

Vad betyder pilen (->) ? Pilen är ett förenklat skrivsätt för `(*data[i]).d1` . Detta betyder att vi hanterar den data som data[i] pekar på och i det här fallet ska vi välja datamedlemmen d1 i posten. Man måste använda parenteser eftersom `'.'` har högre prioritet än `'*'`. Detta klumpiga skrivsätt har man förenklat till `data[i]->d1`.

Sorteringen görs med bubblesort och det är datamedlemmen 'd1' som avgör ordningen:

```
Data *tempAdr;
  for(int pass=0; pass < SIZE-1 ; pass++)
    for( int i=0; i < SIZE-1; i++)
     if(data[i]->d1 > data[i+1]->d1)
      {
         tempAdr = data[i];
         data[i] = data[i+1];
         data[i+1] = tempAdr;
      }
```

Observera att det är pekarna som flyttas om! Därför behövs ett temporärt minnesutrymme för en pekare till Data som används då pekarna byter plats (`Data *tempAdr`).

I följande program hanteras 10 000 poster. En array skapas, värden slumpas och utskrift av de sista 100 posterna görs före och efter sortering. Hela programmet följer på nästa sida:

```
// point_050 Version 10
// En array med pekare till data
// Per Ekeroot 2013-01-13
//-----------------------------------------------------------------------------
#include <iostream>
#include <iomanip>
#include <ctime>
#include <random>
#include <string>
using namespace std;
//-----------------------------------------------------------------------------
struct Data
{
  double d1;
  double d2;
  double d3;
  string s;
};

int main()
{
//-----------------------------------------------------------------------------
//Initiera programmet
//-----------------------------------------------------------------------------
  default_random_engine generator(static_cast<unsigned>(time(0)));
  uniform_real_distribution<double> random(0,100);

  cout << fixed << setprecision(2);
  const size_t SIZE = 10000;

//-----------------------------------------------------------------------------
// Deklarera array med adresser till datatypen Data
//-----------------------------------------------------------------------------
  Data *data[SIZE] = {NULL};

//-----------------------------------------------------------------------------
// Allokera plats för Data och slumpa tal
//-----------------------------------------------------------------------------
  for(size_t i=0; i < SIZE; i++)
  {
    data[i] = new Data;

    data[i]->d1 = random(generator); // Slumpa ett flyttal i intervallet 0 till
    data[i]->d2 = random(generator); // 100 (100 är inte med!)
    data[i]->d3 = random(generator);
    data[i]-> s = "Kalle Svensson i skogen";
  }

//-----------------------------------------------------------------------------
// Skriv ut 100 element osorterat
//-----------------------------------------------------------------------------
  for(size_t i=0 ; i < 100; i++)
    cout << setw(7) << data[i]->d1 <<  setw(7) << data[i]->d2
         << setw(7) << data[i]->d3 <<  setw(25) <<data[i]->s << endl;

    cout << "Press ENTER to continue!";
    cin.get();
```

```
//-------------------------------------------------------------------------
// Sortera
//-------------------------------------------------------------------------
  cout << " ***** Sorting ***** " << endl;
  clock_t start = clock();

  Data *tempAdr;
  for(size_t pass=0; pass < SIZE-1 ; pass++)
    for(size_t i=0; i < SIZE-1; i++)
     if(data[i]->d1 > data[i+1]->d1)
      {
         tempAdr = data[i];
         data[i] = data[i+1];
         data[i+1] = tempAdr;
      }

  cout << " ***** Sorted *****" << endl;
  clock_t stop = clock();
  float t = static_cast<float>(stop - start)/CLOCKS_PER_SEC; // Tillåt sekunder
  cout << " Time = " << t << " [s]" << endl << endl;         // med decimaler!

  cout << "Press ENTER to continue!";
  cin.get();
//
//-------------------------------------------------------------------------
// Skriv ut 100 element sorterat
//-------------------------------------------------------------------------
  for(size_t i=0; i < 100; i++)
    cout << setw(7) << data[i]->d1 <<  setw(7) << data[i]->d2
         << setw(7) << data[i]->d3 <<  setw(25) <<data[i]->s << endl;

//Frigör allokerad plats
  for(size_t i=0; i < SIZE; i++)
           delete data[i];

//
////-------------------------------------------------------------------------
//// Testa med kontinuerlig (vanlig) array
////-------------------------------------------------------------------------
//// Deklarera en array med datatypen Data. Lägg den på heapen.
////-------------------------------------------------------------------------
// Data* data2 = new Data[SIZE];
//
////-------------------------------------------------------------------------
//// Slumpa tal
////-------------------------------------------------------------------------
//
//  for(size_t i=0; i < SIZE; i++)
//  {
//    data2[i].d1 = random(generator); // Slumpa ett flyttal i intervallet 0 till
//    data2[i].d2 = random(generator); // 100 (100 är inte med!)
//    data2[i].d3 = random(generator);
//    data2[i]. s = "Kalle Svensson i skogen";
//  }
////-------------------------------------------------------------------------
//// Skriv ut osorterat
////-------------------------------------------------------------------------
//  for(size_t i= 0 ; i < 100; i++)
//    cout << setw(7) << data2[i].d1 <<  setw(7) << data2[i].d2
//         << setw(7) << data2[i].d3 <<  setw(25) <<data2[i].s << endl;
//
//    cout << "Press ENTER to continue!";
//    cin.get();
```

```
////-------------------------------------------------------------------------
//// Sortera
////-------------------------------------------------------------------------
//  cout << " ***** Sorting ***** " << endl;
//  clock_t start = clock();
//
//         Data temp;
//  for(size_t pass=0; pass < SIZE-1 ; pass++)
//   for( size_t i=0; i < SIZE -1; i++)
//     if(data2[i].d1 > data2[i+1].d1)
//     {
//        temp     = data2[i];
//        data2[i] = data2[i+1];
//        data2[i+1] = temp;
//     }
//
//         cout << " ***** Sorted *****" << endl;
//  clock_t stop = clock();
//  cout << " Tid = " << (stop-start)/CLOCKS_PER_SEC << " [s]" << endl << endl; //

//  cout << "Press ENTER to continue!";
//  cin.get();
////-------------------------------------------------------------------------
//// Skriv ut sorterat
////-------------------------------------------------------------------------
//  for(size_t i= 0 ; i < 100; i++)
//    cout << setw(7) << data2[i].d1 <<  setw(7) << data2[i].d2
//         << setw(7) << data2[i].d3 <<  setw(25) <<data2[i].s << endl;
//
//  delete [] data2; // Frigör allokerad plats

  cout << "Press ENTER to continue!";
  cin.get();

  return 0;
}
```

Kommentarer:

- clock() m.fl. används för att mäta tiden för sorteringen
- I programmets andra del görs en jämförelse med en kontinuerlig array.
- Här kan man skriva `data2[i].d1` eftersom man inte har pekare till data
- I sorteringen av den kontinuerliga arrayen flyttar man **hela posterna** vilket är resurskrävande, särskilt om man har stora poster.