3.3 Array som objekt

Ordet *array* betyder i engelskan *ordnad skara* eller *ordnad uppställning* (*battle array* = stridsordning). Som datalogisk term hittar man i litteraturen begreppen *fält*, *vektor*, *matris*, *lista*, … . Ibland används även *härledd datatyp* som syftar åt att den är baserad på en *annan* datatyp. Vi kommer att använda den enkla termen *array*.

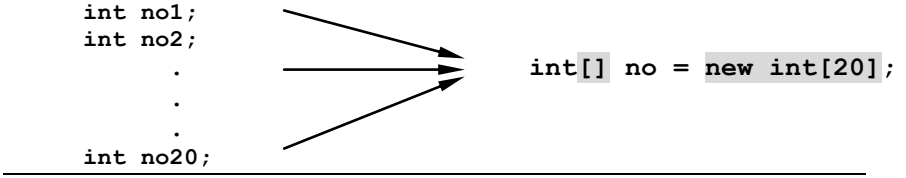
En **array** är en datastruktur, en ordnad mängd av variabler av *samma* datatyp grupperade under *samma* namn.

En array består av ett antal **element** vars *position* kallas för **index**.

Index är synonym till nummer och specificerar varje elements position i arrayen för att ”adressera” elementet. Elementen kan i sin tur vara av enkel, sammansatt eller av referenstyp. Så man kan även – med hjälp av referenser – gruppera objekt till en array. En array är den enklast tänkbara sammansatta datatypen. Som exempel tar vi en array som är sammansatt av den enkla datatypen **`int`**. Varje element i en sådan array kan betraktas som en indexerad dvs numrerad variabel av typ **`int`**.

Hittills behövde vi skriva 20 satser för att definiera 20 heltalsvariabler. Men nu ger array oss möjligheten att göra samma sak med endast *en* sats:

Hittills: enkel datatyp **`int`**: **Nu:** sammansatt datatyp ”*array av* **`int`**”:



Vi definierar *en* variabel **`no`** av datatypen **`int[]`**, använder **`new`** och lägger till informationen om antalet element inom hakparentes: **`[20]`**. Men vad är **`int[]`** för datatyp? Det reserverade ordet **`new`** avslöjar att det är ett *objekt*. **`new`** allokerar minnesutrymme för ett objekt bestående av 20 **`int`**-värden och returnerar den sammanhängande ”minneskedjans” adress – närmare bestämt adressen till dess första cell – till referensvariabeln **`no`**. Därmed har vi att göra med en *referenstyp*: Datatypen **`int[]`** är en *referens till en* **`int`**-*array* som i själva verket är ett *objekt*. För att göra det ännu tydligare kan man skriva den nya koden även i två separata satser:

```
int[] no;
no = new int[20];
```

Det är inte den första utan den andra satsen, närmare bestämt koden **`new int[20]`** som *skapar* själva arrayen. Därför står också storleken **`20`** där det behövs, nämligen i satsen där **`new`** allokerar minne. Typiskt för array är *hakparenteserna* **`[ ]`**, på engelska *brackets*. I satserna ovan har **`[ ]`** två olika betydelser: I den första satsen specificerar **`int[]`** variabeln **`no`**:s *datatyp* som en referens till en **`int`**-array, i den

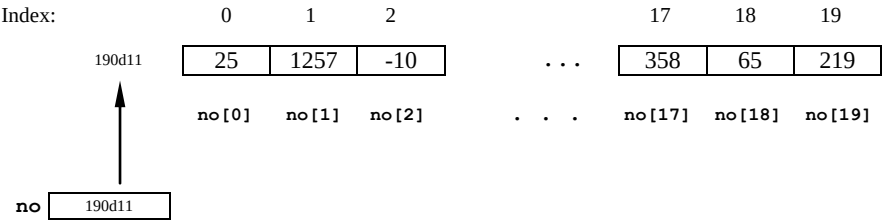
andra satsen innehåller **[20]** arrayens *storlek*. Referensvariabeln **no** ersätter de 20 vanliga **int**-variablerna **no1**, **no2**, …, **no20**, vilket medför en stor effektivitet i koden. Tänk dig att det är inte **20** utan fler data vi vill jobba med. **no** pekar fysiskt på det *första* elementet av arrayen som allokeras i ett sammanhängande minnesutrymme. Därför kan man komma åt de *andra* elementen via *indexering* som är bara ett annat namn för numrering.

Indexering i en array

Låt oss anknyta till exemplet ovan där arrayen och dess referens **no** definieras:

```
int[] no = new int[20];
```

Låt oss ytterligare anta att vissa *värden* – de som visas i bilden nedan – har tilldelats arrayens element efter satsen ovan. Eftersom elementen lagras i ett sammanhängande minnesområde uppstår följande minnesbild av arrayen i datorns RAM:



Medan själva arrayens allokering (den övre delen) görs av **new int[20]**, allokeras minnescellen **no** (den undre delen) av **int[] no**. Kopplingen mellan dem görs av tilldelningsoperatorn, vilket gör att arrayens adress (t.ex. 190d11 – ett hexadecimalt tal) som **new** har genererat, hamnar i minnescellen **no**. Den så uppkomna situationen innebär att **no** *pekar på* eller *refererar till* arrayen. Under arrayens minnesceller har vi skrivit C#-kod som kommer åt varje elements värde: **no[0]** ger den första minnescellens värde 25 som har index **0**, **no[1]** ger den andra minnescellens värde 1257 som har index **1** osv. **no[0]** lagras vid adressen till arrayens första minnescell. **no[1]** lagras vid adressen till den andra minnescellen som ligger **1** x 4 bytes – storleken för en **int** – längre bort från **no**. **no[2]** lagras vid adressen som ligger **2** x 4 bytes längre bort från **no** osv. Adressering i RAM sker nämligen bytevis, så att bytes som är grannar till varandra, har adresser som skiljer sig på *en* enhet. Avgörande för denna indexeringsteknik är att en array alltid allokeras i ett *sammanhängande* minnesområde. Ser man på det hela ur hårdvarans synpunkt kan man förstå varför indexnumreringen börjar med **0** och inte med **1**: **no[0]** kan tolkas som den *adress* som ligger **0** x 4 bytes längre bort från **no**, dvs **no[0]**:s adress är identisk med adressen **no**. Därför gäller:

> Indexregeln: I en array börjar numreringen av index alltid med 0.
> Därför gäller: elementets position = index + 1

Med *position* menas numret som *människan* använder för att numrera elementen. Människor är vana vid att påbörja numreringen av saker och ting med 1. Med *index*

menas numret som *datorn* använder för samma sak. C# och de flesta andra programmeringsspråken börjar numreringen av index i en array med **0**. Tillämpad på exemplet: Det 1:a elementet i den array som **no** refererar till har *värdet* 25 och *index* 0: Positionen är 1 medan indexet är **0**. Det 2:a elementet (värdet 1257) har index **1** och koden **no[1]**, det 3:e elementet (värdet –10) har index **2** och koden **no[2]** osv. Det n:e elementet har alltid index n-1. Därför har också det 20:e elementet (värdet 219) index **19**.

Det är avgörande när man arbetar med array och är samtidigt felkälla nr 1 – om man glömmer det – att hålla isär det mänskliga sättet att numrera som börjar med 1 från C#-kodens sätt som börjar med 0. I exemplet ovan har vi definierat en array av 20 heltalselement med referenserna **no[0], ..., no[19]**. Antalet element är 20. Indexen däremot går från **0** till **19**. Felkälla nr 2 är att förväxla en arrayelements *index* med dess *värde*: Det sista elementet i exemplet ovan har index **19**, men värdet 219. Man har alltid med *två* tal att göra, index (position) och värde (innehåll). Det gäller att hålla isär positionen från innehållet.

Tre egenskaper skiljer objekt från array:

- Indexering
- Allokering i ett sammanhängande minnesområde
- Alla arrayelement har samma datatyp.

Annars behandlas array i C# som objekt: Båda måste skapas med **new** och man kan komma åt båda endast med referensvariabler. Båda initieras till defaultvärden även om de kan förekomma som lokala variabler i metoder.

Definition och initiering av en array

Här testas allt vi sagt hittills om array speciellt indexregeln. Utöver det visas ytterligare en egenskap hos array som relaterar den till objekt, nämligen en datamedlem **Length** som lagrar arrayens storlek när den skapas. Programmet demonstrerar vad som händer om man överskrider arrayens maximala index: Man kan kompilera, men exekveringen stoppas vid överskridningen av indexgränsen, ett tecken på att arrayens minnesallokering sker vid *run time*, dvs programmet körs.

```
// ArrayObj.cs
// Definierar en array som objekt, visar default-initierings-
// värdena 0, tilldelar och skriver ut de nya värden
// Skriver ut arrayens storlek med datamedlemmmen Length
// Överskridning av arrayens index leder till exekveringsfel
using System;
class ArrayObj
{
   static void Main()
   {
     int[] no;                  // Deklarerar en referens no
                                // till en array av int
                                // typ array vars adress
```

```
     no = new int[4];          // new skapar ett objekt av
                               // tilldelas referensen no
//   int[] no = new int[4];    // Alternativt i EN sats

     Console.WriteLine("\n\tArray-storleken:\t\t"+no.Length);
     Console.Write("\n\tArrayens default-initiering:\t");
     foreach (int element in no)
        Console.Write(element + "\t");
     no[0] = 64;               // Tilldelar 1:a elementet
     no[1] = 86;               // värdet 64 osv. Överskriver
     no[2] = 34;               // default-initieringen
     no[3] = -6;
     Console.Write("\n\n\tArrayen efter tilldelning:\t");
     foreach (int element in no)
        Console.Write(element + "\t");
     Console.WriteLine(
       "\n\n\tÖverskridning av arrayens index leder till " +
       "programavbrott:\n\n\t\tno[4] inte definierad\n\t"  +
       "\tIndex 4 överskrider gränsen: Exekveringsfel!")   ;
     no[4] = 1;                // no[4] kan kompileras, men
  }                            // leder till exekveringsfel
}
```

Inte alla satser i programmet **ArrayObj** exekveras. Det blir avbrott när den kompilerade koden **no[4]** i allra sista satsen ska exekveras där index **4** överstiger arrayens tillåtna maximala indexgräns som är **3** därför att **new** i början av programmet allokerar endast **4** minnesceller åt arrayen, nämligen de med index **0**, **1**, **2** och **3**. Någon minnescell med index **4** är inte allokerad. Därför kan vi inte heller referera till den. Men eftersom arrayens allokering sker med **new** och därmed under exekveringstid leder detta till exekveringsfel, medan kompilatorn godtar den syntaxmässigt korrekta koden **no[4]**. Programmet **ArrayObj** ger följande utskrift när den körs:

```
   Arrayens storlek:             4

   Arrayens default-initiering:  0       0       0       0

   Arrayen efter tilldelning:    64      86      34      -6

   Överskridning av arrayens index leder till programavbrott:

            no[4] inte definierad
            Index 4 överskrider gränsen: Exekveringsfel!

Unhandled Exception: System.IndexOutOfRangeException: Index
was outside the bounds of the array.
...
```

Default-initiering av array

Det är anmärkningsvärt att det som gäller för referensen **no** – att den är oinitierad när den skapas – inte gäller för själva arrayen. Referensen **no** är oinitierad och måste initieras explicit eftersom den är en lokal variabel i **Main()**. Men trots att även arrayen är lokal i **Main()** initieras dess element till **0** som är defaultvärdet till datamedlemmar av datatypen **int**. Detta visar att arrayen behandlas som ett objekt. Programmet **ArrayObj** skriver ut arrayelementens värden en gång *innan* och en andra gång *efter* att de har fått värdena **64**, **86**, **34** och **-6**. Generellt gäller:

I C# måste alla lokala variabler i en metod initieras innan de används.
Datamedlemmar i ett objekt initieras automatiskt till default-värden.
Att arrayelementen initieras till 0 (default) visar att arrayen är ett objekt.

En annan slutsats från utskriften av programmet **ArrayObj** är:

Att referera till icke-definierade element i en array leder till exekveringsfel.

C#-kompilatorn kontrollerar inte en arrays indexgränser: **ArrayObj** leder inte till kompileringsfel. Däremot kontrollerar C#-interpretatorn *(C# Virtual Machine)* indexgränserna och tillåter inte åtkomsten till icke-allokerade minnesplatser, dvs stoppar skräpvärden. Detta är ur datasäkerhetssynpunkt är en fördel. Programmen blir stabilare. C++ har i detta avseende en mer liberal attityd. Där ligger ansvaret för kontroll av indexgränserna helt och hållet hos programmeraren.

Att **no[4]** inte är definierat, fast talet **4** ”förekommer” i definitionssatsen **new int[4]**, beror på att **4** i hakparentesen av **no[4]** betyder *index*, medan **4** i **new int[4]** betyder *storlek*. Den korrekta tolkningen av **[]** beror på sammanhanget. **[]** är symbolen för tre olika operatorer som överlagrar varandra dvs betyder olika i olika sammanhang, se sid 102.

foreach-satsen

Denna sats som används i programmet **ArrayObj** (sid 99) är en ny kontrollstruktur som inte kunde tas upp i kapitlet om kontrollstrukturer *(Progr1)* därför att den förutsätter array-begreppet eller liknande sammansatta datatyper, som vi inte hade hunnit gå igenom då.

foreach-satsen är idealisk för att skriva ut sammansatta datatypers värden. Den gör samma sak som **for**-satsen, men har en lite annorlunda – ja t.o.m. lite enklare syntax, om man är förtrogen med arrays. I programmet **ArrayObj** (sid 99) ser satsen ut så här:

```
foreach (int element in no)
    Console.Write(element + "\t");
```

Översatt till svenska:

För varje **element** av arrayen **no**
Skriv ut elementet följt av en tabulator.

element – ett namn som är valt av oss – kallas för **foreach**-satsens *iterationsvariabel*. Den definieras till **int** och motsvarar **for**-satsens räknare. **element** pekar på värdet (innehållet) som står i arrayen. Iteration betyder upprepning och innebär här att satsens kropp upprepas: Programflödet fortskrider från element till element tills alla element är genomgångna. Det reserverade ordet **in** betyder *av* eller *element av*. **no** pekar på arrayen som ska loopas igenom. Därför: ” För varje **element** *av* arrayen **no**”.

foreach-satsens enkelhet består i att den till skillnad från **for**-satsen varken behöver ett start-, steg- eller slutvärde resp. avslutningsvillkor. Den går helt enkelt igenom arrayens *alla* element, från det första till det sista. Det är själva arrayen som bestämmer start-, steg- och slutvärdena. Variabeln **element** pekar i varje varv av loopen på resp. arrayelementets värde och kan sedan användas i loopens kropp för att göra det man önskar. I vårt exempel för att skriva ut arrayens element följt av en tabulator.
foreach-satsens iterationsvariabel måste ha samma datatyp som arrayelementen eller en sådan datatyp som arrayelementens datatyp automatiskt kan konverteras till. I vårt exempel har vi **int**. Det är t.o.m. möjligt att ha egendefinierade datatyper dvs klasser. Ett exempel på det är programmet **ArrayOfRef** (sid 107). Där deklareras iterationsvariabeln i en **foreach**-sats till den egendefinierade klassen **Fish** (sid 106), för att skriva ut ett **Fish**-objekts sort, vikt, längd, pris och frakt.

En viktig egenskap av iterationsvariabeln är att den inte kan ändra arrayelementens värden i **foreach**-satsens kropp. Den är så att säga *read only*. I praktiken innebär detta att iterationsvariabeln inte får förekomma till vänster om tilldelningsoperatorn (**=**) i någon sats i **foreach**-satsens kropp. Vill man i **foreach**-satsens kropp ändra på arrayelementens värden måste man använda **for**-satsen istället med arrayens index som räknare.

Hakparentesernas tre olika betydelser

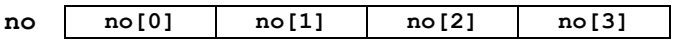
1. **[] som storleksoperator** omsluter i definitioner med **new** *antalet* element i arrayen specificerar därmed arrayens *storlek*. T.ex. innebär koden

```
new int[4]
```

i programmet **ArrayObj** att **new** skapar en array av **int** med **4** element dvs att **4** minnesceller reserveras för lagring av **int**-värden. Det gemensamma för alla dessa element är att de lagras en efter den andra vid adressen eller referensen **no**:

Här är frågan om ”Hur många element?”. I matematiken kallas det *kardinaltal*.

2. **`[]` som indexeringsoperator** omslutar *indexet* till varje element av en array. Här handlar det om ett elements *position* i arrayen. Man anger index inom hakparenteser för att referera till elementet när man vill hämta eller tilldela det ett värde. Indexregeln (sid 98) tillämpas enligt vilken indexeringen börjar med `0`. Därför är `no[4]` i arrayen ovan inte definierat:



Här är frågan om ”Vilket element?”. I matematiken kallas detta *ordinaltal*.

3. **`[]` som en del av datatypen** ”referens till array” omsluter ingenting utan är tom och skrivs direkt efter en datatyp för att definiera en ny referenstyp. T.ex. innebär satsen

```
int[] no;
```

i programmet `ArrayObj` att en minnescell allokeras (en referensvariabel med namnet `no` definieras) för lagring av en adress till en `int`-array. Vi kan i fortsättningen använda namnet `no` för att komma åt arrayen vid denna adress. I satsen ovan har referensen `no` inte initierats. Det sker inte heller automatiskt, för `no` är en lokal variabel i `Main()`. Det sker först med tilldelningen `no = new int[4];` som initierar referensen explicit.