Kapitel 8

Arrays

8.1 Vad är en array ?

När vi pratade om loopar sade vi så här (sid 105):

> ”Datorn har några egenskaper som är helt överlägsna motsvarande egenskaper hos människan: snabbheten, noggrannheten och förmågan att effektivt lagra och hantera stora datamängder samt förmågan att inte bli trött.”

Kontrollstrukturen *repetition* i sina olika varianter: **`while`**-, **`for`**- och **`do`**-satser utnyttjar både snabbheten och förmågan att inte bli trött. Detta ska nu kompletteras med ett verktyg som även utnyttjar datorns andra överlägsna egenskap: Att kunna effektivt lagra och hantera *stora datamängder*. Detta verktyg är datatypen *array* som i engelskan ordagrant betyder *ordnad uppställning* (*battle array* = stridsordning), en ordnad skara av värden i datorns minne. Andra beteckningar som används i litteraturen är *fält*, *vektor*, *matris*, *lista*. Vi kommer att använda *array*.

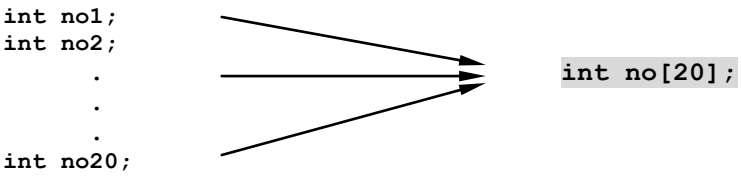
En array är en ordnad mängd av variabler av *samma* datatyp grupperade under *samma* namn och lagrade i ett *sammanhängande* minnesområde. En array består av ett antal element. Elementens position i arrayen kallas för index. Indexnumreringen börjar med 0, inte med 1.

Den sammanhängande lagringen underlättar adresseringen av elementen via indexet och är möjligt därför att alla element är av samma datatyp. Kända datatyper har hittills varit de enkla datatyperna (sid 75). En array är inte längre en enkel utan en s.k. *sammansatt* datatyp, den enklast tänkbara sammansatta datatypen.

En *sammansatt datatyp* representerar fler än ett värde åt gången, t.ex. flera heltal, flera flyttal, flera tecken, flera sanningsvärden osv. Det finns i litteraturen även det synonyma begreppet *härledd datatyp* dvs baserad på en annan datatyp. Man kan gruppera *enkla* datatyper till den nya härledda datatypen array.

Om vi t.ex. grupperar variabler av den enkla datatypen **`int`** får vi den sammansatta datayp som i kod skrivs **`int[]`** och läses *”array av* **`int`***”*. Varje element i en sådan array kan betraktas som en indexerad dvs numrerad variabel av typ **`int`**. Anta att vi vill deklarera 20 heltalsvariabler, då bjuder array på följande förenkling:

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



Hittills har vi skrivit *20* satser (koden till vänster) för att deklarera 20 **int**-variabler. Men nu med array som sammansatt datatyp har vi möjligheten att göra samma sak med endast *en* sats genom att deklarera *en enda* variabel – visserligen inte längre en vanlig variabel utan en *arrayvariabel* – och lägga till informationen om antalet element i den. På så sätt har vi använt den nya sammansatta datatypen *array av* **int** för att definiera vår arrayvariabel **no**. Typiskt för den nya datatypen *array av* **int** är hakparenteserna **[ ]** i deklarationssatsen ovan som innehåller *antalet* element i arrayen. Arrayvariabeln **no** ersätter de 20 vanliga variablerna **no1**, **no2**, …, **no20** och består nu i sin tur av 20 *element*. Varje element är en heltalsvariabel som kan lagra ett heltalsvärde. Enda skillnaden är *sättet* dvs *koden* att komma åt dessa värden. Det är namnet på arrayvariabeln samt index till resp. element som måste anges för att komma åt värdet i elementet. Indexet är ett nummer som specificerar varje elements position i arrayen.

Låt oss anknyta till exemplet ovan där vi hade definitionssatsen: `int no[20];` som reserverar 20 minnesceller för lagring av 20 värden av typ **int**. Låt oss ytterligare anta att värdena i bilden nedan tilldelats alla element i arrayen **no**. De lagras i ett sammanhängande minnesområde. Det uppstår följande:

Minnesbild av arrayen no:

25	1257	-10	. . .	358	65	219
no[0]	**no[1]**	**no[2]**	. . .	**no[17]**	**no[18]**	**no[19]**

Under minnescellerna står hur C++ adresserar minnescellerna i en array och på vilket sätt man i koden kommer åt varje element. Exemplet visar hur indexeringen av element i en array görs. Anmärkningsvärt är att indexnumreringen börjar med **0**. Medan vi människor är vana vid att påbörja numreringen av ett antal objekt med 1, gör C++ det inte när det gäller numreringen av element i en array. Varför kommer vi att förstå när vi behandlar sambandet mellan pekare och array (sid 201). Tills vi når dit måste vi acceptera följande indexregel där vi med position menar numret som människan använder för att numrera elementen.

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

Tillämpad på exemplet: Det 1:a elementet i arrayen **no** ovan (värdet 25) har index 0: Positionen är 1 medan indexet är 0. C++ kodar det med **no[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 gäller att hålla isär det mänskliga sättet att numrera som börjar med 1 från C++ kodens sätt som börjar med 0. Vi har definierat 20 heltalsvariabler **no[0], ..., no[19]**. Antalet element är 20. Indexen går från 0 till 19.

8.2 Definition och initiering av en array

Följande program testar allt vi hittills sagt om arrays speciellt indexregeln. Det visar skillnader mellan vanliga variabler och arrays samtidigt som det avslöjar att de för arrays så typiska hakparenteserna **[ ]** inte alltid har samma betydelse.

```
// ArrayDef.cpp
// Definierar en array av int och initierar den elementvis
// For-sats skriver ut värdena med resp. index
// For-satsens räknare används samtidigt som arrayens index
// Ingen kontroll av indexgränserna i array via kompilatorn
#include <iostream>
using namespace std;

int main()
{
   int no[4];                          // Definition av en array
   no[0] = 64;                         // Elementvis initiering
   no[1] = 86;                         // av arrayen
   no[2] = 34;
   no[3] = -6;
   for (int i=0; i <= 3; i++)          // i = index = räknare
      cout << "\n\tArrayens "<< i + 1 << ":a element no["<< i
           << "] har värdet "<< no[i] << " och index " << i
           << "\n";
   cout << "\tIndex -1 och 4 ligger utanför och är "
        << "ej definierade.\n\n\tResultat:\n";
   cout << "\tno[-1] = " << no[-1] << " med index -1 och\n"
        << "\tno[ 4] = "  << no[4]  << " med index  4 "
        << "är odefinierade skräpvärden!" <<"\n";
}
```

Det blir enklare att prata om programmet om man även tittar på en körning:

```
Arrayens 1:a element no[0] har värdet 64 och index 0

Arrayens 2:a element no[1] har värdet 86 och index 1

Arrayens 3:a element no[2] har värdet 34 och index 2

Arrayens 4:a element no[3] har värdet -6 och index 3

Index -1 och 4 ligger utanför och är ej definierade.

Resultat:
no[-1] = -858993460 med index -1 och
no[ 4] = -858993460 med index  4 är odefinierade skräpvärden!
```

Körningen ovan visar att icke-definierade arrayelement varken leder till kompilerings- eller exekveringsfel. Index **-1** och **4** överskrider de definierade indexgränserna **0** och **3**, den ena till vänster, den andra till höger. Arrayelementen **no[-1]**

och **no[4]** är varken definierade eller tilldelade några värden. Ändå kan man kompilera och köra programmet utan något felmeddelande. Inte ens en varning påpekar att man använt kod som är odefinierad. Anledningen är följande:

> I en array kontrollerar C++ kompilatorn inte indexgränserna utan endast arraynamnet.

Ett annat namn än det definierade arraynamnet **no** leder till kompileringsfel. Däremot kan vi använda vilket index som helst, även om det överskrider de definierade gränserna, utan att det blir kompileringsfel. Ansvaret för kontroll av indexgränserna ligger helt och hållet hos programmeraren. Av indexregeln (sid 160) följer att negativa index generellt inte är tillåtna, även om kompilatorn inte protesterar. Skälet för denna liberala attityd är bl.a. strävan efter snabb kompilering, vilket förstås är på bekostnad av säkerheten.

Man kan ju undra varför **no[4]** inte är definierat – som vi hävdar ovan – fast det ”förekommer” i definitionssatsen **int no[4];** Det ser bara ut *som om* det förekommer. Detta beror på att hakparenteserna **[]** inte har samma betydelse i programmets alla satser. Den korrekta tolkningen av **[]** beror på sammanhanget:

Hakparentesernas två olika betydelser

1. **I definitionssatser** omsluter hakparenteserna *antalet* element i arrayen dvs arrayens *storlek*. T.ex. innebär raden

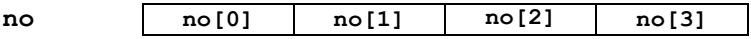
   ```
   int no[4];          // Definition av en array
   ```

   i programmet **ArrayDef** att variabeln **no** definieras till datatypen *array av* **int** med **4** element dvs att **4** minnesceller reserveras för lagring av **int**-värden. Det gemensamma för alla dessa element är arraynamnet **no**:

   Här är frågan om ”Hur många element?”. Detta kallas för *kardinaltal*.

2. **I alla andra satser** omslutar hakparenteserna *index* 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 160) tillämpas enligt vilken indexeringen börjar med **0**. Därför är t.ex. **no[4]** i arrayen **no** inte definierat:

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

Den ovan beskrivna skillnaden i tolkningen av **[]** har viktiga praktiska konsekvenser som vi kommer att se senare.

Initieringslista

Precis som det finns skillnader i definitionen av arrayvariabler jämfört med vanliga variabler, finns även skillnader vid initieringen dvs första tilldelningen. T.ex. är initieringen av arrayen **no** i programexemplet **ArrayDef** (sid 161) – en sats för varje element – inte särskilt lämplig för arrays, speciellt om man skulle tillämpa samma teknik på större arrays. Men just hanteringen av stora datamängder var ju motiveringen för att syssla med array. Kan man inte effektivisera initieringen? Jo, till en viss gräns. Det finns i huvudsak två möjligheter: antingen att använda **for**-satser eller att slå ihop definitionen med tilldelningen till en kortform som använder sig av en s.k. *initieringslista*. Båda har vi använt i följande program:

```
// ArrayInit.cpp
// Kortform för definition och initiering av en array i EN
// och samma sats med hjälp av en initieringslista
// Elementvis tilldelning av en array kan med fördel göras
// med en for-sats: Arrayens index = for-satsens räknare
#include <iostream>
using namespace std;

int main()
{
   int no[] = { 64, 86, 34, -6 }; // Kortform för definition
                                  // och initiering i EN sats
                                  // med initieringslista
   int copy[4];                   // Endast definition
   for (int i=0; i <= 3; i++)     // Elementvis initiering med
       copy[i] = no[i];           // for-sats. Direkt (array-
                                  // vis) initiering copy=no;
                                  // skulle ge kompileringsfel
   cout << '\n';
   for (int i=0; i <= 3; i++)
       cout << "\tcopy:s "<< i + 1 << ":a element copy[" << i
            << "] = " << copy[i] << " med index " << i
                                                  << "\n\n";
}
```

En körning av programexemplet **ArrayInit** visar att värdena från arrayen **no** verkligen kopierats över till arrayen **copy**:

```
    copy:s 1:a element copy[0] = 64 med index 0

    copy:s 2:a element copy[1] = 86 med index 1

    copy:s 3:a element copy[2] = 34 med index 2

    copy:s 4:a element copy[3] = -6 med index 3
```

Både definitionssatsen och initieringssatserna i **ArrayDef** – det är de 5 första satserna i **main()** – kan slås ihop till den enda satsen:

```
int no[] = { 64, 86, 34, -6 };    // Kortform för definition
                                  // och initiering i EN sats
```

som gör två saker: Först, fram till tilldelningstecknet definieras arrayen **no** *utan* någon uppgift om arrayens storlek. Sedan, från och med tilldelningstecknet tilldelas arrayen **no**:s element fyra värden som står i en kommaseparerad lista grupperad inom klamrarna **{ }** som kallas arrayens *initieringslista*. Satsen ovan är endast en kortform för de fem första satserna i **main()**-funktionen till **ArrayDef** och gör precis samma sak som de. Hakparenteserna i kortformen (före **=**) tillhör definitionsdelen. Därför måste deras innehåll tolkas som arrayens storlek som kan och bör utelämnas eftersom satsen inte är avslutad än efter den avlutande hakparentesen utan kompilatorn kompilerar satsen fram till semikolonet. På så sätt får kompilatorn informationen om arrayens storlek i initieringslistan, räknar alltså antalet element inom klamrarna **{ }**. Det är också tillåtet att explicit ange det korrekta antalet element inom hakparenteserna **[ ]**. Kompilatorn ignorerar denna uppgift om man anger ett tal som är lika med eller större än det faktiska antalet element som finns i initieringslistan. Däremot blir det kompileringsfel om man anger ett tal som är mindre än det faktiska antalet element. För att undvika eventuella komplikationer beroende på felräkning rekommenderas att låta kompilatorn räkna och utelämna uppgiften om arrayens storlek när man definierar och tilldelar en array i en och samma sats. I praktiken är det bäst att i kortformen skriva ett tomt hakparentespar **[ ]** precis som vi gjort i programexemplet **ArrayInit**. Observera att man får använda kortformen ovan endast i samma sats som definitionen.

Däremot är det inte möjligt att utelämna uppgiften om arrayens storlek när man i en sats endast definierar en array som t.ex. i raden:

```
int copy[4];                      // Endast definition
```

eftersom här finns inte någon annan information om arrayens storlek än uppgiften inom hakparenteserna. Utan den informationen kan inget minnesutrymme för arrayen reserveras. Denna array har definierats i **ArrayInit** för att skapa en kopia av arrayen **no** genom att tilldela **no**:s värden till **copy** dvs tilldela värdena av alla element i arrayen **no** till resp. element i arrayen **copy**. Om **no** och **copy** varit vanliga variabler hade man kunnat skriva tilldelningssatsen **copy = no;** då variabeln **no** är både definierad och tilldelad. Även **copy** är definierad. Men med arrayvariabler går det inte. Det ger kompileringsfel om man försöker att tilldela värdena i en array direkt (arrayvis) till en annan array. Följande gäller:

> Tilldelning av värden i en array kan endast göras elementvis.

Den elementvisa tilldelningen kan antingen göras ”manuellt” dvs varje elementtilldelning i en separat sats – exempel på det hade vi i **Array** – eller i en **for**-sats.

Array i en `for`*-sats*

Man kan också effektivisera denna primitiva metod genom att använda en loop, med fördel en **`for`**-sats. Exempel på det har vi i **`ArrayInit`**:

```
for (int i=0; i <= 3; i++)    // Elementvis tilldelning med
    copy[i] = no[i];          // for-sats
```

Här görs en elementtilldelning per varv i **`for`**-satsen. Därför sker tilldelningen också elementvis vilket kroppens kod på den andra raden visar: vanliga variablers tilldelningssats **`copy = no;`** ersätts av arrayvariablers elementvisa tilldelningssats **`copy[i] = no[i];`** där **`i`** som både **`for`**-satsens räknare och arrayernas index går igenom alla element av båda arrayerna.

Att **`for`**-satsen är den mest använda loopen vid hantering av arrays är ingen tillfällighet. Det är naturligt att **`for`**-satsen lämpar sig bäst för arrays då det i förväg kända antalet element i en array ger upphov till det i förväg kända antalet repetitioner i **`for`**-satsen. Den logiska slutsatsen blir att sätta **`for`**-satsens räknare till arrayens index för att gå igenom alla element i arrayen. Därför kan, när en array bearbetas med for-sats, följande teknik användas med fördel:

> Arrayens index = **`for`**-satsens räknare

Denna teknik har även använts vid utskrift av arrayen **`copy`**:s värden i slutet av programmet **`ArrayInit`**.

8.3 Stränghantering med array

I början av detta kapitel motiverade vi behandlingen av arrays med att kunna lagra och bearbeta stora datamängder. Ett exempel på det är *textbehandling* där stora textmängder snarare är regel än undantag. Textbehandling är ett klassiskt område för datortillämpning just pga datorns förmåga att effektivt och snabbt kunna hantera stora datamängder.

I de två föregående avsnitten använde vi endast datatypen ”array av **int**” som är sammansatt av den enkla datatypen **int**. När array kan sätta samman enkla datatyper till en ny, större enhet, kan man använda den även för att behandla text som en sammansättning av tecken. För att representera tecken har vi den enkla datatypen **char**. Vi behöver bara bilda en array av **char** för att representera text. Men vad exakt är text? Det enda vi kan säga är att den består av ett antal tecken. Men var går den övre gränsen? Ett ord, en mening, en rad, en sida, en hel bok, ... – allt detta kan vara text. I det vanliga språket har man *ordet* som den minsta enheten i text och *mellanslaget* som avskiljare mellan två ord. Punkt och komma samt andra specialtecken är ytterligare verktyg för att strukturera text i vanligt språk. Däremot det vanliga språkets definitioner och verktyg lämpar sig inte alltid när man vill representera text i datorn. Man använder begrepp som är mer anpassade till datorns arbetssätt. Ett av dem är begreppet *sträng*.

> Sträng = samling av tecken som avslutas med *nolltecknet.*
> Nolltecknet har ASCII-koden 0 och kan i C++ kodas med escapesekvensen `\0`, med **`NULL`** eller `0`.

Nolltecknet

Självklart har nolltecknet inget att göra med siffran 0 som har ASCII-koden 48. Nolltecknet är det första av de icke-skrivbara, icke-läsbara styr- eller kontrolltecknen i början av ASCII-tabellen (sid 83). Dessa tecken har olika funktioner i olika språk. I C++ har nolltecknet bl.a. funktionen att avsluta strängar. *Sträng* är den minsta enheten i en text och *nolltecknet* avskiljaren mellan två strängar. Kvarstår frågan: När och hur läggs nolltecknet till en sträng? Skriva kan man inte, det är inte skrivbart. Läsa kan man inte heller, det är inte läsbart. Faktum är att nolltecknet automatiskt läggs till i slutet när man sätter strängen inom citationstecken `" "` vilket kännetecknar datatypen sträng. Sedan kan man spåra det med ASCII-koden 0 eller med escapesekvensen **`\0`**. Självklart ska man hantera strängar med datatypen **`string`** som introducerades på sid 71. Generellt är **`string`** stabilare och borde föredras i professionellt sammanhang. Men då kan man inte längre på en elementär nivå manipulera tecknen, vilket vi just vill göra. Dessutom vill vi lära oss mer om arrays och nolltecknet som vi kommer att koda med **`\0`** även om det lika bra går att göra det med **`NULL`** eller **`0`**. Därför definieras här strängar som **`char`**-arrays.

För att definiera och tilldela en **char**-array kan vi använda en initieringslista (sid 163):

```
char letter[] = {'a', 'b', 'c', 'd'};
```

precis som i: `int no [] = {64 , 86 , 34 , -6 };`

Den första satsen skapar variabeln **letter** som en array av **char** och tilldelar till den en sammanhängande följd av tecken. Men **letter** blir inte automatiskt en strängvariabel då det avslutande nolltecknet saknas. I koden finns det ingen information om att denna teckenföljd ska vara en sträng. Därför läggs i det här fallet nolltecknet inte till automatiskt. Men arrayvariabeln **letter** kan ersättas med en strängvariabel om vi i koden lägger till nolltecknet som sista tecknet i arrayen:

```
char text[] = {'a', 'b', 'c', 'd', '\0'};
```

Ett enklare alternativ är:

```
char text[] = "abcd";
```

Citationstecknen som kännetecknar datatypen sträng, visar att **abcd** *skall* vara en sträng och då läggs nolltecknet *automatiskt* till teckenföljden. Vi behöver inte längre göra det explicit. Därför blir **text** en *strängvariabel* som tilldelas *strängkonstanten* **abcd**. Observera att vi har åstadkommit detta med en array av **char** i definitionsdelen kombinerad med citationstecken i tilldelningsdelen.

I alla satser ovan skrivs definition och tilldelning av arrayvariabeln **text** i en och samma sats. Vi gör det även i följande program där lär vi oss stränghantering med array, manipulering med nolltecknet samt strängfunktionen **strcpy()** som kopierar strängar:

```
// ArrayChar.cpp
// Skriver ut en sträng och mäter dess längd med sizeof
// Kopierar (tilldelar) strängen till en ny variabel
// Kapar strängen med nolltecknet
// Definition av strängvariabler till datatypen array av char
#include <iostream>
using namespace std;
int main()
{
   char text[] = "C++";         // Definition och tilldelning
                                // i EN och samma sats
   // char text[] = {'C', '+', '+', '\0'};  // Sträng som ovan
   // char text[] = {'C', '+', '+'};        // Ingen sträng
   for (int i=0; i<=3; i++)
       cout << i+1 <<":a tecknet i \"C++\" är " << text[i] <<
                                                      '\n';
   cout << "\nSträngen \"C++\" tar " << sizeof(text)
        << " bytes: 3 tecken + nolltecknet\n\n"
        << "4:e tecknets ASCII-kod är " << (int) text[3] <<
                                                      '\n'
        << "Nolltecknets ASCII-kod är " << (int) '\0' << '\n'
```

```
        << "Alltså är 4:e tecknet i \"C++\" nolltecknet\n\n";
   char copy[4];
   strcpy_s(copy, text);     // text="C++" kopieras till copy
   copy[1] = '\0';           // Avkortning av "C++" till "C"
   cout << copy << " är en del av " << text << "\n\n";
}
```

Vi kommer att använda även separata definitionssatser. Men än så länge utnyttjar vi kortformen **`char text[] = "C++";`** eftersom den är så bekväm. Har man skrivit dem separat, kan initieringslistan inte användas efteråt för att tilldela arrayen.

Ett körresultat av programmet ovan visar att nolltecknet är osynligt dvs icke-läsbart och icke-skrivbart. Man kan identifiera det endast via ASCII-koden:

```
1:a tecknet i "C++" är C
2:a tecknet i "C++" är +
3:a tecknet i "C++" är +
4:a tecknet i "C++" är

Strängen "C++" tar 4 bytes: 3 tecken + nolltecknet

4:e tecknets ASCII-kod är 0
Nolltecknets ASCII-kod är 0
Alltså är 4:e tecknet i "C++" nolltecknet

C är en del av C++
```

Utskriften visar att ASCII-koden är 0: Det är det första tecknet i ASCII-tabellen och har inte det minsta att göra med siffran 0. En första indikation för nolltecknets existens i arrayen **`text`** är returvärdet **`4`** till **`sizeof(text)`** där strängvariabeln **`text`** har värdet **`C++`** som innehåller 3 tecken. Varje tecken är av typ **`char`** som tar 1 byte i minnesutrymme. Om nolltecknet inte fanns i arrayen **`text`**, skulle **`sizeof`** returnera 3 bytes. Men nolltecknet tar också 1 byte så **`sizeof`** returnerar **`4`** bytes när den tillämpas på **`text`** som parameter. Beviset för nolltecknets existens i arrayen **`text`** är att ASCII-koden till både arrayens 4:e tecken och nolltecknet är 0.

Slutligen använder vi i **`ArrayChar`** nolltecknet för att ändra i strängen **`C++`**. När man har ett tecken som fungerar som strängslutstecken kan man använda det för att manipulera strängar på alla möjliga sätt, t.ex. för att kapa en sträng vilket vi gjorde ovan. Men före detta ingrepp vill vi göra en kopia av strängen, ändra i kopian och på så sätt spara originalet för att sedan kunna använda båda. Kopiering innebär tilldelning av strängen till en ny strängvariabel som definieras i satsen **`char copy[4];`** Men tilldelning av arrayvärdena kan endast göras elementvis vilket i regel görs med en **`for`**-sats. Kommer man tänka på att överföra med **`for`** även nolltecknet när det gäller array av **`char`**? Annars blir kopian ingen sträng. Därför finns den fördefinierade funktionen **`strcpy()`** som står för ***str**ing **copy***. Den hanterar nolltecknet automatiskt och löser dessutom problemet nästan lika bekvämt som

tilldelningstecknet. Den kopierar sin andra parameter till sin första. Båda parametrar måste vara strängar. Så här fungerar **strcpy()**:

Ersätter raden nedan: **strcpy(copy, text);**

Får ej skrivas i C++ kod: **copy = text**

copy ⟵ text

Den mellersta raden går förstås inte att skriva i C++ pga regeln om elementvis tilldelning av arrayvärden (sid 164). Efter kopieringen av strängen **C++** från **text** till **copy** tar vi kopian som nu också innehåller strängen **C++** och gör om den med:

```
copy[1] = '\0';
```

Här använder vi själva nolltecknet i koden för att avsluta strängen **copy** efter eget önskemål. Vi sätter nolltecknet på elementet med index 1 så att strängen **C++** kapas efter bokstaven **C** därför att tecknet **+** i elementet med index 1 (det första **+** tecknet i strängen) skrivs över av nolltecknet. Om vi jämför med originalet har vi:

text	**C**	**+**	**+**	**\0**
copy	**C**	**\0**	**+**	**\0**

Därför får vi **C** när vi skriver ut **copy** och **C++** när vi skriver ut **text**. Vad som står i **copy**:s element efter det första nolltecknet är irrelevant. Visserligen får man återanvända dem pga definitionssatsen **char copy[4];** men de är i princip oinitierade så länge det första nolltecknet finns där och strängvariabeln **copy** har värdet **C**.

När läggs nolltecknet till automatiskt? Sker det vid definitionen eller tilldelningen? Följande program visar att det är vid tilldelningen nolltecknet tillkommer:

```
// NullChar.cpp
// Läser in en sträng som kan även innehålla mellanslag och
// skriver ut strängens alla tecken med tillhörande ASCII-kod
// Använder nolltecknet för att avsluta utskriften i for-
// loopen precis när den inmatade strängen är slut
#include <iostream>
using namespace std;

int main()
{
   char namn[20];            // Reserverar plats för en sträng
                             // med max 20 tecken inkl. \0
   cout << "Ge ditt ditt för- och efternamn: ";
   cin.getline(namn, 20);    // Läser in en sträng med max 20
                             // tecken till strängvariabeln
                             // namn där mellanslag kan ingå
   cout << "\nIndex\t\tTecken\t\tASCII-kod\n"
        << "----------------------------------------\n";
```

```
   int i;                              // for-satsens räknare
   for (i=0; (namn[i] != '\0'); i++) // Skriver ut tecken så
       cout << i << "\t\t" << namn[i]// länge tecknen inte \0
            << "\t\t" << (int) namn[i] << '\n';
   cout << i << "\t\t" << namn[i] << "\t\t"
        << (int) namn[i] << "\n\n";
}
```

Här har vi separerat definitionen av **char**-arrayen från tilldelningen för att kunna mata in vilken sträng som helst. Kortformen i **ArrayChar** hade sina fördelar – enkelheten och kompaktheten – men nackdelen var att tilldelningen var hårdkodad och inte kunde göras interaktivt. Vill vi göra tilldelningen via inmatning kan vi tyvärr inte använda kortformen. Detta leder i sin tur till att vi redan vid definitionen måste bestämma oss hur stor arrayen ska vara. Det går inte att i en separat definition av en array utelämna uppgiften inom hakparenteserna om antalet element. Eftersom vi sedan vill tilldela ett namn – för- och efternamn – till strängvariabeln **namn**, tar vi en storlek som verkar rimlig för namn i allmänhet. Vi antar att de längsta namnen inte kan ha fler än 19 bokstäver. Vi reserverar 1 element också för nolltecknet och definierar därför strängvariabeln **namn** med

```
char namn[20];
```

som en array av **char** med **20** element. En sådan definition av en array kallas *statisk minnesallokering*. Allokera betyder i datasammanhang att reservera minnesutrymme. Definitionen ovan är en statisk minnesallokering därför att den redan vid kompilering reserverar ett sammanhängande minnesområde bestående av **20** minnesceller där varje minnescell kan lagra ett **char**-värde i arrayvariabeln **namn**. Storleken på detta minnesområde är oföränderlig under hela programmet. Arrays som definieras på detta sätt är alltid statiska i den bemärkelsen. Man måste bestämma sig redan vid definitionen, alltså *innan* man använder arrayen, hur stor den ska vara. Detta kan uppfattas som en stor inskränkning och är också i många fall ett verkligt hinder för vissa applikationer. Alternativet – *dynamisk minnesallokering* – är förstås möjligt i C++, men kräver att man använder *pekare* vilket introduceras i kapitel 10. I programmet **NullChar** vill vi läsa in användarens namn och lagra det i arrayen **namn**. Då vi i förväg inte vet hur långt det inmatade namnet kommer att vara, måste vi för säkerhets skull reservera lite mer plats än vad som kommer att behövas i de flesta körningar av programmet. Ett begränsat slöseri med minnesutrymme är oundvikligt när man använder statisk minnesallokering.

Stränginmatning med `cin.getline()`

Första frågan som dyker upp i **NullChar** är: Varför har vi inte använt en vanlig **cin**-sats för att läsa in användarens namn? Svaret är: därför att **cin** tolkar mellanslaget som avskiljare mellan två värden som ska läsas in. Alla s.k. *vita tecken* dvs **Enter**, mellanslag och tabulator tolkas av **cin** som avskiljare vid inmatning (sid 63). Därför skulle en **cin**-sats endast läsa in förnamnet och ignorera efternamnet när användaren matar in båda namnen skilda med mellanslag. Men vi vill ju läsa in

både för- och efternamn och lagra det hela som en sträng i arrayen **namn**. Därför använder vi en s.k. *metod* [*] eller *medlemsfunktion* som är fördefinierad i **cin** som heter **getline()** och anropas med s.k. *punktnotation* (sid 224):

```
cin.getline(namn, 20);
```

Denna funktion tolkar inte de vita tecknen som avskiljare vid inmatning så att strängar som läses in med **cin.getline()** även kan inkludera mellanslag och tabulatorer. Man kan alltså läsa in längre texter med **cin.getline()** – som det engelska namnet antyder en hel rad – där ”rad” betyder fram till *radslutstecknet* dvs **Enter**-tangenten. Funktionen **cin.getline()** sätter nolltecknet till den inlästa strängen och lagrar den inkl. nolltecknet i sin första parameter, i exemplet i arrayen **namn**. Observera att **cin.getline()** har två obligatoriska parametrar. Den första är strängvariabeln som man vill lagra den inmatade strängen i. Den andra är det maximala antalet tecken inkl. nolltecknet som strängen kan innehålla.

Efter att ha läst in både för- och efternamn till strängvariabeln **namn** skrivs **namn**:s alla tecken med tillhörande index och ASCII-koder ut i en **for**-sats vars villkor

```
namn[i] != '\0'
```

formuleras med hjälp av nolltecknet för att kunna avsluta loopen precis när strängen tar slut. Vilkoret innebär: Så länge tecknet med index **i** inte är nolltecknet ska loopen fortsätta. Så går loopen igenom strängen tecken för tecken då index **i** som samtidigt är **for**-satsens räknare börjar med **0** och ökar med **1** för varje varv. När denna genomgång påträffar nolltecknet avslutas loopen – utan att någon som helst information om den inmatade strängens längd har behövt användas. Så här kan en körning se ut:

```
Ge ditt ditt för- och efternamn: Kalle Karlsson

Index           Tecken          ASCII-kod
-----------------------------------------
0               K               75
1               a               97
2               l               108
3               l               108
4               e               101
5                               32
6               K               75
7               a               97
8               r               114
9               l               108
10              s               115
```

[*] Funktioner definierade i en klass heter *metoder*. De kan endast anropas med *punktnotation* dvs deras anrop står efter en punkt. Före punkten står i regel ett *objekt* – ett exemplar av den klass där metoden är definierad.

```
11              s               115
12              o               111
13              n               110
14                              0
```

Med hjälp av detta resultat ska vi nu förklara följande frågor:

1. När exakt läggs nolltecknet till arrayen?
2. När exakt slutar **for**-satsen?
3. Vad händer med de arrayelement som definierats men inte utnyttjas?

1. När man matar in för- och efternamn och trycker på **Enter**, läggs nolltecknet till arrayen direkt efter den sista bokstaven av efternamn, i körexemplet ovan efter bokstaven **n** med index **13**. Allra sista raden i utskriften ovan visar det osynliga nolltecknet med index **14** och ASCII-koden **0**. Definitionen reserverar alltid – vid varje körning – **20** tecken, men nolltecknet läggs till i slutet av varje inmatad sträng – dvs på olika ställen vid olika inmatningar. Närmare bestämt är det den funktion som läser in strängen dvs **cin.getline()** som lägger till nolltecknet så snart man trycker på **Enter** vid inmatning. Nolltecknet läggs till alltså vid inläsningen dvs tilldelningen och inte vid definitionen. Detta gäller generellt.

2. Det sista varvet i **for**-satsen skriver ut:

```
13          n               110
```

dvs indexet **i** är **13**, tecknet **n** är skilt från **\0** – därför skrivs det ut – samt tillhörande ASCII-kod som är **110**. Sedan blir **i 14** pga **i++**. Tecknet med detta index dvs **namn[14]** är lika med **\0**. Därför blir **for**-satsens villkor falskt och loopen avslutas med värdet **14** för **i**. Observera att det sista **i** som skrivs ut är **13**, men att **i** är **14** när vi lämnar **for**-satsen beroende på att **i++** utförs *efter* **cout**-satsen. Se **for**-satsens flödesplan på sid 110.

3. Efter **for**-satsen använder vi i en **cout**-sats räknaren **i** utan att ge den ett nytt värde. Dvs **i**:s värde efter **for**-satsen är samma som vi lämnade den med, nämligen **14**. Från punkt **2** vet vi också att tecknet med detta index dvs **namn[14]** är lika med **\0**. Därför skrivs ut:

```
14                              0
```

dvs det osynliga nolltecknet med ASCII-koden 0. Att **for**-satsens räknare **i** är väl definierad även efter **for**-satsen beror på att den är definierad som variabel *före* **for**-satsen, som en slags global variabel – global relativ till **for**-satsen. Det har vi gjort just för att kunna använda den efteråt för att visa att nolltecknet finns på index **14**, den aktuellt tilldelade strängens sista plats och inte på index **19**, den definierade strängvariabelns sista plats. Vid varje kör-

ning kommer denna plats vara olika pga olika inmatningar. Att nolltecknet finns på index `14` och inte på index `19` (arrayens sista reserverade element) visar att nolltecknet lagts till vid tilldelningen och inte vid definitionen.

Det inmatade namnet var kortare än `20` tecken. Vilka värden står i de resterande 5 minnescellerna efter nolltecknet? Även de har definitionen reserverat plats för utan att den aktuella körningen utnyttjat dessa platser. Med dem händer samma sak som alltid med variabler som är definierade, men inte tilldelade: De förblir oinitierade och innehåller skräp, vilket lätt kan testas genom att låta programmet skriva ut deras skräpvärden.

8.4 Kryptering av text

Vi ska nu dra lite praktisk nytta av våra samlade kunskaper om bl.a. slumptal, ASCII-koder, array, stränghantering, funktioner och referensanrop, för att med ganska enkla medel skriva en liten applikation om kryptering av text. Följande program läser in en text, skickar den till en externlagrad funktion som kan både kryptera och återställa texten med ett slumptal som krypteringsnyckel. Metoden som används för kryptering är väldigt enkel, nästan primitiv, men kan lätt ersättas av mer sofistikerade algoritmer.

```
// EncryptTest.cpp
// Skickar en text som array samt en krypteringsnyckel till
// funktionen krypt() där den krypteras. Krypteringsnyckeln
// slumpas fram vid varje körning tidsberoende. Referens-
// anrop gör den krypterade texten tillgänglig i main()
// krypt() anropas en 2:a gång med den krypterade texten och
// inverterad (neg.) krypteringsnyckel för att återställa den
#include <iostream>
#include <ctime>
using namespace std;
#include "Encrypt.h"                // Innehåller krypt()
#include "..\MyRand.h"              // Innehåller myRand()
                                    // se sid 146
int main()
{
   srand(time(0));                 // Gör slumpen tidsberoende
   char text[80];
   cout << "\nGe en text:\t\t";
   cin.getline(text, 80);
   int key = myRand(1, 1000);      // Slump-krypteringsnyckel

   krypt(text, key);               // 1:a anropet krypterar

   cout << "\nKrypterad  text:\t" << text << "\n\n";

   krypt(text, -key);              // 2:a anropet återställer
                                   // med negativ nyckel
   cout << "Återställd text:\t" << text << "\n\n"
        << "Krypteringsnyckeln:\t" << key << "\n\n";
}
```

Med en array av **char** allokeras minne för texten med en maximal längd på **80** tecken. För att kunna inkludra mellanslag i texten, läses den in med **cin.getline()** och lagras i arrayvariabeln **text**. Med ett första anrop av funktionen **krypt()**:

```
krypt(text, key);
```

som är externlagrad i filen **Encrypt.h**, överförs parametern **text** med referensanrop då den är definierad som array. Därför är ändringarna som görs med **text** i

funktionen **krypt()**, tillgängliga efter anropet. Texten är okrypterad före och krypterad efter detta anrop både i fukntionen och i **main()**. Den andra parametern **key** däremot överförs med värdeanrop då den är definierad till den enkla datatypen **int**. Efter **krypt()**:s första anrop skrivs den krypterade texten ut. Sedan anropas **krypt()** med **-key**, det negativa värdet av **key**, för att återställa texten vilken sedan skrivs ut för kontroll. Hur krypteringsnyckeln **key** och därmed hela krypteringsmetoden fungerar, förstår man bäst om man samtidigt tittar på funktionen **krypt()**:

```
// Encrypt.h
// Tar emot en text via arrayen t och krypterar den genom att
// förskjuta alla tecken med n steg i ASCII-tabellen
// Kontrollerar textens slut med nolltecknet

void krypt(char t[], int n)
{
   for (int i=0; t[i] != '\0'; i++)
       t[i] = t[i] + n;
}
```

Krypteringsmetoden är väldigt enkel: tecknens ASCII-värden ökas med **n** i satsen **t[i] = t[i] + n;** genom vanlig addition. Att det verkligen adderas **n** till *ASCII-koden* till **t[i]** beror på att **t[i]** är av typ **char** och att en teckenvariabel i aritmetiska uttryck tolkas som sin ASCII-kod – ett tal man kan räkna med (sid 85). **for**-satsen som går igenom hela strängen genom att koppla räknaren till arrayens index, gör att hela texten förskjuts med **n** steg i ASCII-tabellen. **n** får sitt värde genom kopiering (värdeanrop) från **key** vid första och **–key** vid andra anropet. **key**:s värde i sin tur slumpas fram i **main()** med hjälp av funktionen **myRand()** vars kod finns på sid 146. Detta värde som är något heltal mellan **1** och **1000** skickas som krypteringsnyckel till **krypt()** via den andra parametern. Vid andra anropet av **krypt()** skickas **–key** för att återställa texten. Genom att ersätta **t[i] + n** med mer sofistikerade formler kan man utveckla mer avancerade krypteringsalgoritmer. Filen **MyRand.h** som innehåller funktionen **myRand()** inkluderas i **EncryptTest** med **#include "..\MyRand.h"**. Anledningen till **..** är att filen är placerad i mappen strax över den aktuella mappen för att använda den till andra program.

Programmet **EncryptTest** kan köras på olika sätt. Varje körning ger en annan slumpmässig krypteringsnyckel. Här ett exempel på en körning:

```
Ge en text:              abcd

Krypterad  text:         ÁÂÀ©

Återställd text:         abcd

Krypteringsnyckeln:      340
```

Man kan kontrollera krypteringen för hand: Man ser att bokstaven **a** förskjutits till **Á**. Krypteringsnyckeln har vid denna körning varit **340**. ASCII-koden till **a** som är 97, har förskjutits **340** steg vidare till 97 + 340 = 437 som överskrider datatypen **char**:s övre gräns. Det blir *overflow:* Värdet måste räknas *modulo* 2^m där *m* är antalet bitar i minnesutrymme som står till förfogande för den aktuella datatypen. Pga datatypen **char** är *m* = 1 byte dvs 8 bitar. Overflow-värdet 437 måste alltså räknas *modulo* 2^8 där 2^8 = 256. Men 437 *modulo* 256 är 118 dvs resten vid heltalsdivision av 437 med 256 är 118 eller: 437 % 256 = 118. På sid 114 kan man se att 118 är ASCII-koden till tecknet **Á**. Därför har **a** förskjutits till **Á** med krypteringsnyckeln **340**.

Självklart borde i en skarp applikation krypteringsnyckeln inte skrivas ut utan endast sparas i variabeln **key** för att använda den vid återställningen. Vi gör det här endast för experimentens skul.

Då krypteringsnyckeln är ett slumptal mellan **1** och **1000** som tack vare programsatsen **srand(time(0));** varierar från körning till körning, får vi en annan kryptering vid en annan körning även om vi matar in samma sträng. Prova gärna! En körning med en längre text som även innehåller mellanslag, ger:

```
Ge en text:          Längre text som skulle kunna stå i en fil

Krypterad  text:
É╚▓½Â®d©®╝©dÀ|▒dÀ»┤▒▒®d»┤▓▓ÑdÀ©╩d¡d®▓d¬¡▒

Återställd text:     Längre text som skulle kunna stå i en fil

Krypteringsnyckeln: 580
```

Lägger man till filhantering i programmet **EncryptTest** kan samma funktion **krypt()** användas för kryptering av filer. Vi kommer att göra det i nästa kapitel som tar upp filhantering.

Men först ska vi avsluta detta kapitel med ett sista avnsnitt som behandlar en utvidgning av arraybegreppet: en array vars element i sin tur är arrays.