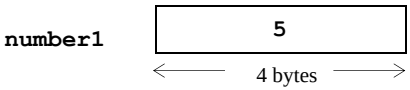
Kapitel 10

Pekare

10.1 Vad är en pekare ?

När vi deklarerade och initierade variabler sa vi så här (sid 81):

> ”Vad händer t.ex. i satsen **`int number1 = 5;`** ?
>
> **1. *Minnesallokering*** Minnescell reserveras i datorns RAM för lagring av **`int`**-värden. Namnet på denna minnescell blir **`number1`** …
>
> 
>
>
> **`...`**
>
> **3. *Adressering*** har med namngivning att göra. Programmets *logiska* variabelnamn **`number1`** kopplas till minnescellens *fysiska* adress i RAM. Det görs för att komma åt minnescellen genom att referera till variabelnamnet. Variabler gör minnescellerna i hårdvaran åtkomliga för mjukvaran.”

Vid deklaration av en variabel uppstår alltså en länk mellan en *adress* (hårdvara) och det variabelnamn vi använder i koden (mjukvara). När vi pratar om *adress* menar vi alltid en fysisk plats i datorns RAM-minne (*Random Access Memory*).

Pekare är en ny datatyp för lagring av *adresser* till värden som kan vara av vilken vanlig datatyp som helst.

Till varje vanlig datatyp finns en pekardatatyp: *pekare-till*-datatyp

Variabler deklarerade till denna nya datatyp kallas *pekarvariabler*.

Exempel: Adressen till en **`int`** lagras i en pekarvariabel av typ *pekare-till-***`int`**.

När ett C++ program körs sker all minneshantering – datorns verkliga jobb – med de fysiska adresserna. Varje översättning av ett variabelnamn till en fysisk minnesadress och vice versa tar en viss tid. Därför är det effektivare att kunna arbeta med adresser direkt.

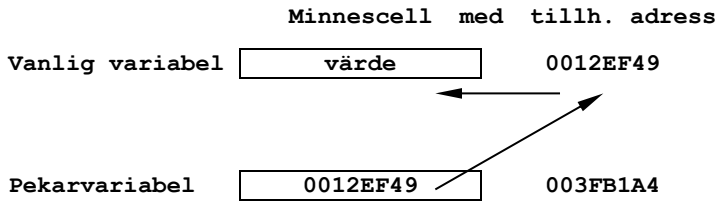
Ju närmare man kan komma hårdvaran desto snabbare kan datorn bearbeta data. C++ är p.g.a. sina rötter i C det programmeringsspråk bland de objektorienterade högnivåspråken som står hårdvaran närmast. Språket har flera verktyg för att kontrollera och styra hårdvaran. Ett av dessa verktyg är *pekare* vars ursprung går tillbaka till C. C++ har ärvt pekarkonceptet från C.

Pekare = ett nytt parallellt system

Alla våra program som vi hittills skrivit, kan skrivas om till nya pekarversioner om man bara byter ut alla vanliga variabler till pekarvariabler. Man refererar till de minnesceller som lagrar programmets variabler med adresser istället för variabelnamn. Det verkar som om vi har tagit steget in i en helt ny värld inom C++ programmering. I själva verket har vi satt på oss nya glasögon som ser på samma värld ur ett mer maskinnära perspektiv. Närmare bestämt är pekare inte *en* datatyp utan täcker C++ språkets alla datatyper. Till varje vanlig datatyp finns en pekardatatyp.

Med pekarvariabler kan man göra allt man kan göra med vanliga variabler, och mer därtill. Det speciella med pekarvariabler är att deras värden är adresser. Bilden nedan illustrerar hur man arbetar med pekare:

En pekarvariabels värde = adressen till en vanlig variabel



Språkbruket bland programmerare:

”Pekarvariabeln *pekar på* den vanliga variabeln”

Man tilldelar en pekarvariabel en vanlig variabels adress för att komma åt den vanliga varibelns minnescell indirekt via adressen istället för via variabelnamnet.

Detta förklarar samtidigt ordet *pekare*. Själva pekarvariabelns adress – adressens adress så att säga – är i detta sammanhang av mindre praktisk betydelse, men den finns där i alla fall.

Pekare tillåter programmeraren att från koden komma åt minnesadresser vilket öppnar helt nya möjligheter att skriva program. I ett program utan pekare görs all minnesallokering vid kompileringen, dvs *statiskt*. Med pekare finns möjligheten att allokera minne under programmets gång, dvs *dynamiskt*. Vid statisk minnesallokering måste man i förväg reservera utrymme som sedan kanske inte används i full utsträckning vid körning vilket innebär slöseri med minnesutrymme. Vid dynamisk minnesallokering reserveras utrymme när det behövs under exekveringen. På så sätt kan man skräddarsy användningen av minnesresurserna.

10.2 Deklaration och initiering av en pekare

Följande program simulerar situationen ”*pekar på*” som beskrevs ovan och introducerar ***** som symbolen för pekare och *adressoperatorn* **&** (tecknet *ampersand*) som symbol för *adressen* till en variabel. Det ger oss möjligheten att från koden komma åt vanliga variablers adresser utan att behöva använda det speciella format som finns för adresser, nämligen hexadecimala tal. Detta görs genom att deklarera en pekarvariabel och tilldela till den adressen till en vanlig variabel.

```
// Pointer.cpp
// Deklaration och initiering av en pekarvariabel
// Asterisken * är symbolen för pekarvariabel
// Amperand & framför en variabel ger adressen till variabeln
#include <iostream>
using namespace std;

int main()
{
   double vanligVar = 15.6;
   double *pekarVar;       // Deklarerar en pekare-till-double

   pekarVar = &vanligVar; // Initierar pekarVar med adressen
                          // till vanligVar
                          // "pekarVar pekar på vanligVar"
   cout << "\n\t\t\tVärde\t\t\tAdress\t\t\tStorlek\n\n"
        << "Vanlig variabel:\t" << vanligVar      << "\t\t\t"
        << &vanligVar << '\t'   << sizeof(vanligVar)<< "\n\n"
        << "Pekarvariabel:\t\t" << pekarVar          << "\t"
        << &pekarVar << "\t"    << sizeof(pekarVar)  << "\n";
 }
```

Programmet simulerar minnesbilden på förra sidan när det körs:

```
                    Värde               Adress            Storlek

Vanlig variabel:    15.6                000000E592CFF        8

Pekarvariabel:      000000E592CFFB58    000000E592CFFB78     8
```

Som man ser är pekarvariabeln **pekarVar**:s *värde* en adress som är den vanliga variabeln **vanligVar**:s *adress*, vilket är ett resultat av tilldelningssatsen **pekarVar = &vanligVar;**. Men låt oss börja med deklarationssatsen:

```
double *pekarVar;
```

Satsen deklarerar variabeln **pekarVar** till datatypen pekare-till-**double**. Det är asterisken ***** som talar om för kompilatorn att den variabel som följer dvs **pekarVar** ska vara en pekarvariabel. Observera att själva pekarvariabelns namn är **pe-**

karVar *utan* asterisk. Placeringen av ***** kan variera så att deklarationen kan även se ut så här:

double* pekarVar; eller **double * pekarVar;**

Eventuella mellanslag mellan datatyp och variabelnamn saknar betydelse. Avgörande är att asterisken ***** står *mellan* dem för att kännetecknа variabeln som följer som en pekarvariabel. Fördelen med notationen (sättet att skriva) **double *pekarVar;** som vi kommer att använda i fortsättningen, är att man ser att det handlar om en pekarvariabel, fördelen med **double* pekarVar;** (OBS! skillnaden) är att man ser att det handlar om datatypen pekare-till-**double**. Generellt är:

datatyp *****

C++ syntax för datatypen *pekare-till-datatyp* som t.ex. kan användas vid explicit typkonvertering (ex. på nästa sida). Det finns inga begränsningar för val av *datatyp* som t.o.m. i sin tur kan vara en pekarvariabel vilket leder till datatypen *pekare-till-pekare*. Vill man deklarera *flera* pekarvariabler av samma typ i *en* sats måste asterisken upprepas i den kommaseparerade listan, t.ex.:

```
double *pekarVar1, *pekarVar2, *pekarVar3, ... ;
```

Utelämnas ***** på den andra, tredje eller någon efterföljande plats i listan kommer dessa variabler bli deklarerade som vanliga **double**-variabler och inte som pekarvariabler.

Precis som hos vanliga variabler reserverar deklarationssatsen **double *pekarVar;** minnesutrymme av den storlek som är föreskriven för datatypen pekare-till-**double**. Observera att med denna sats reserveras minnesutrymme endast för själva pekarvariabeln, men inte för det den ska peka på, dvs endast för adressen, inte för det **double**-värde som ska lagras *vid* denna adress. I programmet **Pointer** har vi med hjälp av **sizeof(pekarVar)** fått denna information. Körresultatet på förra sidan visar att pekarvariabelns minnesstorlek är **4** bytes. Pekarvariabler till *alla* datatyper tar alltid lika mycket i minnesutrymme som datatypen **int** gör. I den aktuella C++ installation vi använder är det **4** bytes. Det beror på att värdet som lagras i en pekarvariabel, är en adress oavsett vilken typ av data den kommer att peka på. Adresser representeras i C++ i ett visst format nämligen med *hexadecimala tal*. Körresultatet ovan visar att pekarvariabeln **pekarVar**:s värde är **0012FF78**. Alla hexidecimala tal till en viss övre gräns kan lagras i **4** bytes. Det anmärkningsvärda är att pekare alltid tar **4** bytes oavsett hur stora datamängder de än pekar på.

Hexadecimala tal

Det är tal som är representerade i talsystemet med basen 16. Om vi hade 8 fingrar på varje hand vore denna anmärkning onödig för då kunde vi räkna med hexadecimala tal. Det avgörande skälet för att människan bestämt sig för att räkna med decimala tal är antalet fingrar – inte för att det decimala systemet i sig är det mest effektiva. Ett bevis på det är datorns binära talsystem. När det gäller minnesadresser har man i C++ som i de flesta andra språk bestämt sig för att *visa* adresser i

hexadecimal form. Anledningen är bl.a. för att lätt känna igen dem som adresser och bättre kunna skilja dem från andra datatyper. Närmare bestämt är det **cout** som visar adresser i detta format vid utskrift.

Om du tar fram Kalkylatorn i Windows (fotnot sid 122) kan du omvandla tal till och från olika talsystem bl.a. mellan **Dec**imal och **Hex**adecimal. Glöm inte att i menyn Visa välja välja undermenyn Avancerad. Om man väljer Hex (längst till vänster under displayen) aktiveras en knapprad med bokstäverna A-F längst ned till höger. Dessa bokstäver används nämligen som ”siffror” i det hexadecimala talsystemet. Precis som man har de 10 siffrorna 0-9 i det decimala talsystem som bygger på basen 10, använder sig det hexadecimala talsystemet av de 16 siffrorna 0-9 och A-F där A simulerar den hexadecimala ”siffran” 10, B 11 osv. och F är motsvarigheten till 15. Därför förekommer i hexadecimala tal även bokstäverna A-F som t.ex. i pekarvariabeln **pekarVar**:s värde **0012FF78** enligt körresultat av programmet **Pointer**. I Windows Kalkylator kan man omvandla detta hexadecimala tal till dess decimala motsvarighet: 1 245 048. I äldre C++ installationer kan man få utskriften **0x0012FF78** istället, där det inledande prefixet **0x** är kännetecknet för hexadecimala tal. Enligt den aktuella C++ standarden skriver **cout** ut hexadecimala tal utan prefix. Men vill man i koden tilldela pekarvariabeln **pekarVar** adressen ovan direkt utan adressoperator, måste prefixet sättas. De två inledande nollorna kan ignoreras då de inte bidrar med någon information till talet:

```
pekarVar = (double *) 0x12FF78;
```

Observera att en explicit typkonvertering till datatypen pekare-till-**double** behövs. Utan den blir det kompileringsfel vilket visar att man måste specificera vilken datatyp pekaren ska peka på. Utan denna specificering är hexadecimala tal i sig varken av typ pekare-till-**double** eller kan automatiskt konverteras till den, vilket än en gång understryker att det inte finns någon datatyp *pekare* utan endast *pekare-**till** någon datatyp*. När vi pratar om pekare menar vi alltid pekare-till någon datatyp.

Men vi ska inte fördjupa oss i hexadecimala tal. Det är inte heller nödvändigt då det finns ett bekvämt sätt att komma åt adresser till minnesceller som redan är allokerade t.ex. genom deklaration. I C++ finns en fördefinierad operator för just detta ändamål nämligen adressoperatorn som vi redan lärde känna i programmet **Pointer** och som vi ska närmare titta på i nästa avsnitt där vi samtidigt kommer att lära oss hur man kommer åt *värdet* till en vanlig variabel med hjälp av en pekare som pekar på den.

10.3 Adress- och värdeoperatorn

Programmet **Pointer**:s ”svaghet” ur pekarsynpunkt är att tilldelningen och utskriften av variabeln **vanligVar** sker på konventionellt sätt – med den vanliga variabelns namn – och inte med pekarvariabeln som pekar på den. Kopplingen mellan den vanliga och pekarvariabeln är enkelriktad: Vi får adressen till ett värde, men inte omvänt: värdet lagrat vid adressen.

För att helt och hållet gå över till kod som endast använder pekarvariabler, måste vi konstruera en dubbelriktad koppling. Vi måste ha *två* operatorer som t.ex. i matematiken **+** och **–** eller ***** och **/**. Dessa operatorer uppträder parvis, är motsatta till varandra och tar ut varandra. Man kallar dem för *inversa* operatorer. Även i C++ finns faktiskt *två* operatorer varav den ena ger *adressen* till ett värde och den andra *värdet* lagrat vid en adress. Därför kallar vi dem för *adress-* och *värdeoperator*. Den första har vi använt i programmet **Pointer** och den andra kommer vi att visa i följande program:

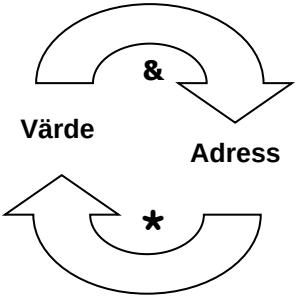
```
// Value.cpp
// Att referera till en variabel indirekt via dess adress
// istället för konventionellt via variabelnamnet.
// * i är icke-deklarationssatser värdeoperatorn som refere-
// rar till variabelns värde som pekarvariabeln pekar på.
// Den är invers till adressoperatorn & .
#include <iostream>
using namespace std;

int main()
{
   double vanligVar;           // Deklaration utan initiering
   double *pekarVar = &vanligVar; // pekarVar pekar på van-
                                  //                   ligVar
   *pekarVar = 15.6;           // Här är * värdeoperatorn
                               // VÄRDET i den variabel som
                               // pekarVar pekar på, dvs van-
                               // ligVar, ska bli 15.6
   cout << "\n\t\t\tVärde\t\t\tAdress\t\t\tStorlek\n\n"
        << "Vanlig variabel:\t" << vanligVar << "\t\t\t"
        << &vanligVar << '\t' << sizeof(vanligVar) << "\n\n"
        << "Pekarvariabel:\t\t" << pekarVar << "\t"
        << &pekarVar << "\t" << sizeof(pekarVar) << "\n\n"
        << "Värde- * och adressoperatorn & är inversa"
        << " operatorer:\n\n\t\t"
        <<       " *(&vanligVar) = " << *(&vanligVar)
        << "\n\t\t &(*pekarVar) = " << &(*pekarVar)<< "\n\n";
}
```

Inittieringen av **vanligVar** sker med pekaren: ***pekarVar = 15.6;** Asterisken ***** har här en annan betydelse än symbolen för pekare, när den deklareras. Här betyder asterisken *värdet* i den minnescell med adressen **pekarVar**.

Ett körexempel av programmet **Value** ger följande bild:

```
                    Värde                Adress             Storlek

Vanlig variabel:    15.6                 000000087731FB38     8

Pekarvariabel:      000000087731FB38     000000087731FB58     8

Võrdeoperatorn * och adressoperatorn & õr inversa operatorer:

                   *(&vanligVar) = 15.6
                   &(*pekarVar)  = 000000087731FB38
```

De två sista raderna bekräftar att adress- och värdeoperatorn är inversa till varandra. Adressoperatorn går från värdet **vanligVar** till adressen, medan värdeoperatorn går från adressen **pekarVar** till värdet. Utför man dem efter varandra återvänder man till utgångspunkten. Cirkeln i bilden till höger sluts. De tar ur varandra. T.ex. betyder:




```
*(&vanligVar)
```

värdet som lagras vid adressen som anges inom parentes, dvs **vanligVar**:s värde. ***** och **&** neutraliserar varandra och vi får tillbaka **vanligVar** vars värde **15.6** skrivs ut i körresultatet. Analogt betyder:

```
&(*pekarVar)
```

adressen till variabeln som anges inom parentes. Där står variabeln som **pekarVar** pekar på, dvs **vanligVar**. I sin helhet får vi adressen till **vanligVar** som är **pekarVar**. Igen: **&** och ***** tar ut varandra och vi får tillbaka den ursprungliga variabeln **pekarVar** vars värde **0012FF78** skrivs ut i körresultatet.

Anledningen till att sambandet mellan dessa operatorer tas upp här, är att det belyser och underlättar förståelsen för pekaren. Man kan memorera bättre, när man känner igen en underliggande struktur – något som bakom det rent tekniska ger ett koncept och en mening till det hela. Därför sammanfattar vi:

> Värdeoperatorn * är invers (motsatt) till adressoperatorn &

En annan intressant aspekt, som även har praktisk betydelse för kodningen, är att den lilla asterisken ***** som ganska ofta förekommer i våra program med pekare, faktiskt har olika betydelser i olika sammanhang. Vi ska titta närmare på denna skillnad.

Asteriskens två olika betydelser

1. **I deklarationssatser** talar asterisken **`*`** om för kompilatorn att den variabel som följer ska vara en pekarvariabel. Följande sats betyder:

   ```
   double *pekarVar;
   ```

   att variabeln **`pekarVar`** deklarareras till datatypen pekare-till-**`double`**. Samtidigt reserveras en minnescell på **`4`** bytes för lagring av adresser till **`double`**-värden.
   I explicita typomvandlingar: (*datatyp* **`*`**) *uttryck* betyder **`*`** att *uttryck* ska konverteras till datatypen *pekare-till-datatyp*.

2. **I alla andra satser** har asterisken **`*`** en helt annan betydelse (bortsett från multiplikation). När den i sådana satser skrivs framför en redan deklarerad och tilldelad pekarvariabel, refererar den till den variabelns *värde* som pekarvariabeln pekar på:

   ```
   *pekarVar = 15.6;
   ```

   betyder att *värdet* i den variabeln som **`pekarVar`** pekar på ska bli **`15.6`**. I **`Value`** är variabeln som pekaren **`pekarVar`** pekar på, pga den andra satsen i **`main()`**, den vanliga variabeln **`vanligVar`**. Alltså tilldelar satsen ovan variabeln **`vanligVar`** värdet **`15.6`**. Denna tilldelning sker indirekt dvs via adressen istället för konventionellt med variabelnamnet.

Vi ska nu behandla de inversa operatorerna adress- och värdeoperatorn:

Adressoperatorn &

I programmet **`Pointer`** gör deklarationen av variabeln **`vanligVar`** som **`double`** att en minnescell allokeras av storleken 8 bytes för lagring av ett **`double`**-värde. Därmed är minnescellens adress skapad och i princip känd för programmet. Tack vare *adressoperatorn* är denna adress även tillgänglig för programmet. Adressoperatorn symboliseras med tecknet & och tillämpas på en variabel genom att skriva:

`&variabelnamn` eller **`& variabelnamn`**

Även här saknar mellanslaget betydelse. Avgörande är att tecknet **`&`** skrivs *framför* variabelnamnet. Då returnerar adressoperatorn adressen till denna variabel. Variabeln behöver inte ens vara tilldelat ett värde. Det räcker att den är deklarerad. Även om vi i **`Pointer`** inte hade tilldelat variabeln **`vanligVar`** värdet **`15.6`** hade vi kunnat skriva ut dess adress med **`cout << &vanligVar;`** direkt efter deklarationen. Variabeln som adressoperatorn tillämpas på behöver inte vara en vanlig variabel. Den kan även vara en pekarvariabel. Då får man pekarvariabelns adress. Vi har i **`Pointer`** tillämpat adressoperatorn både på en vanlig variabel och en pekarvariabel. Körresultatet visar att pekarvariabelns *adress* är en annan adress än pekarvariabelns *värde*. Adressoperatorn kan även tillämpas på namngivna konstanter, däremot inte på icke-namngivna konstanter.

Vi återkommer till den mest intressanta observationen i körresultatet av **Pointer**, nämligen att pekarvariabeln **pekarVar**:s *värde* som är en adress, är identiskt med den vanliga variabeln **vanligVar**:s *adress* vilket beror på tilldelningssatsen:

```
pekarVar = &vanligVar;
```

som gör att **pekarVar** pekar på **vanligVar**. Vi kan rent formellt göra denna tilldelning då adressoperatorn returnerar adressen till en **double** och pekarvariabeln **pekarVar** är deklarerad som en pekare-till-**double**. Datatyperna på båda sidor överensstämmer alltså. Hade, om detta inte varit fallet, automatisk typkonvertering tillämpats? Svaret är nej. Observera att alla regler för *automatisk typkonvertering* endast gäller för *enkla* datatyper. Pekare är inga enkla datatyper utan sammansatta eller härledda då de är baserade på andra datatyper. Kom ihåg att med pekare allid menas pekare-till någon datatyp. Så, försöker man att tilldela **pekarVar** en annan datatyp än pekare-till-**double** som t.ex.: **pekarVar = 2.5;** blir det kompileringsfel med felmeddelandet att ett **double**-värde (enkel datatyp) inte kan konverteras till en pekare-till-**double** (sammansatt).

Med tilldelningssatsen ovan har vi gjort något man typiskt brukar göra med pekare. Vi har tagit en befintlig minnesadress, nämligen **&vanligVar**, och skrivit den i pekarvariabeln **pekarVar**:s minnescell. Vi har kopplat ihop den vanliga variabeln med pekarvariabeln. Först nu förtjänar pekaren sitt namn då den pekar på ett annat allokerat minnesutrymme, den vanliga variabelns minnesutrymme. Gör man inte det är pekaren inte väl definierad. Kom ihåg att först när en variabel är väl definierad får den användas. Först när en pekarvariabel pekar på ett minnesutrymme får den användas. Den typiska ”användningen” av en pekarvariabel är att med hjälp av den komma åt det minnesutrymme som den pekar på. Detta har vi inte gjort i **Pointer** men kommer att göra i nästa programexempel.

Värdeoperatorn *****

Vi kallar den så med tanke på att den returnerar *värdet* analogt till adressoperatorn som returnerar adressen. *Värdeoperatorn* ***** (asterisken i en annan betydelse än hittills) ger oss möjligheten att komma åt vanliga variablers *värden* via deras adresser efter att ha kopplat ihop den vanliga variabeln med pekarvariabeln. Andra beteckningar som förekommer i litteraturen är ***-operatorn**, *indirektoperatorn* eller på engelska *dereference operator*.

Nästa programexempel **Value** på nästa sidan introducerar värdeoperatorn genom att visa hur man kommer åt *värdet* till en vanlig variabel med hjälp av en pekare som pekar på den. Samtidigt demonstreras sambandet mellan adress- och värdeoperatorn. I programmet **Value** har till skillnad från **Pointer** variabeln **vanligVar** inte tilldelats ett värde direkt via sitt namn utan indirekt via sin adress dvs en pekare som pekar på den. Dessutom har **pekarVar**:s deklarations- och tilldelningssats slagits ihop:

```
double *pekarVar = &vanligVar;
```

där i deklarationsdelen (till vänster om tilldelningstecknet) asterisken ***** använts som symbol för pekarvariabel. Precis som hos vanliga variabler kan man initiera pekarvariabler redan vid deklarationen. Gör man det till en vana kan man minska risken för oinitierade skräpadresser.

Helt ny här är tilldelningssatsen **`*pekarVar = 15.6;`** där asterisken ***** inte längre är kännetecknet för pekarvariabel. Satsen tilldelar istället den variabel som **`pekarVar`** pekar på, nämligen **`vanligVar`**, värdet **`15.6`**. Detta beror på att asterisken ***** inte har samma betydelse i programmets alla satser. Vi är redan vana att använda samma tecken för olika saker. Symbolen för multiplikation är samma som för pekare vid deklaration, och nu kommer en användning till för asterisken. Tillgången till tecken är ju begränsad, så det finns helt enkelt inte ett unikt tecken till varje tänkbar operation. Sammanhanget måste avgöra den rätta tolkningen.

Dessutom kan programmet **`Value`** användas – om man vill – för en kraschtest. Tar man bort kopplingen mellan den vanliga och pekarvariabeln, dvs deklarerar bara **`pekarVar`** utan att tilldela den adressen till **`vanligVar`**, så kan man väl kompilera koden. Men exekveringen leder till minneskrasch därför att det uppstår en pekare med en oinitierad skräpadress som pekar på ett minnesutrymme som med största sannolikhet är upptaget av ett annat program i datorns RAM.

Inversa operatorer

Inversa operatorer finns fler än man tror: Addition och subtraktion, multiplikation och division, att potentiera och dra roten, i C++: ökningsoperatorn **`++`** och minskningsoperatorn **`--`** , den logiska negationen **`!`** , **`new`** och **`delete`** (sid 284) osv. Tillämpar man en operator på en operand och sedan den inversa operatorn på resultatet av denna operation får man tillbaka den ursprungliga operanden. Därför säger man: Inversa operatorer tar ut varandra, de neutraliserar varandra. T.ex. tillämpar man operationen +3 på operanden 4 och sedan den inversa operationen –3 på resultatet, får man tillbaka operanden 4. Dvs: (4 + 3) – 3 = 4. Ett annat enkelt exempel är (6 x 5) / 5 = 6. Den logiska negationen är ett exempel på en operator som är sin *egen invers*: **`!(!v) = v`** där **`v`** kan vara ett godtyckligt villkor eller en godtycklig utsaga. Man kan alltså förstå den dubbla negationen även med hjälp av den inversa operatorn. Vad har allt detta att göra med adresser och pekare? Jo, även värdeoperatorn kan man förstå utifrån adressoperatorn som vi redan känner till, plus konceptet om den inversa operatorn.

Vad gör adressoperatorn? Den tar en variabel och returnerar dess adress. Följaktligen gör värdeoperatorn som dess invers det motsatta: den tar en adress och returnerar dess variabel. Men vad är ”dess variabel”? Jo, det är inget annat än den variabel pekaren pekar på. Lite mer noggrant måste vi alltså formulera: Värdeoperatorn tar en pekare som pekar på en variabel och returnerar denna variabel. Själva variabeln som returneras behöver endast vara deklarerad. Antingen är den redan tilldelad ett värde, då returnerar värdeoperatorn detta värde. Eller så är variabeln inte tilldelad än, då kan man med värdeoperatorn tilldela den ett värde. Detta gjorde vi i programmet **`Value`** med satsen **`*pekarVar = 15.6;`** där ***** är värdeoperatorn. När vi

skriver denna sats pekar pekarvariabeln **pekarVar** redan på den vanliga variabeln **vanligVar** genom tilldelningen **pekarVar = &vanligVar;** vilket gör att **vanligVar** i resten av programmet blir utbytbar mot ***pekarVar**. De refererar båda till samma minnescell och kan användas vare sig för att läsa eller skriva data. Med ***pekarVar = 15.6;** skrivs värdet **15.6** i minnescellen. Med **cout << vanligVar;** läses det från minnescellen. Beviset är att vi får **15.6** utskrivet med denna **cout**-sats fast vi tilldelat variabeln med ***pekarVar**. Detta visas i utskriften till programmet **Value** som visades tidigare.

Vi återkommer till det tidigare utlovade kraschtestet. Kommenterar man i **Value** bort kopplingen **pekarVar = &vanligVar;** mellan den vanliga variabeln och pekarvariabeln kan man fortfarande kompilera, men det blir exekveringsfel följt av minneskrasch. Testet kan lätt göras genom att ändra den andra satsen i **main()** till:

```
double *pekarVar;          // = &vanligVar;
```

Kopplingen som bryts ovan är ju inget annat än pekarvariabeln **pekarVar**:s tilldelningssats. Tar man bort den blir pekaren oinitierad. Den är deklarerad, men pekar på ingenting, närmare bestämt på inget minnesutrymme allokerat av programmet. Men som vi vet finns det ju rent fysiskt inga tomma minnesceller. I den deklarerade pekarvariabeln **pekarVar**:s minnescell står från tidigare användning något slumpmässigt odefinierat skräpvärde – en skräpadress. Med värdeoperatorn ***pekarVar** tillämpad på pekarvariabeln försöker vi nu komma åt värdet som **pekarVar** pekar på. Pga skräpadressen pekar **pekarVar** på ett minnesutrymme som med största sannolikhet är upptaget av ett annat program i datorn. Försöker man att från ett program komma åt det minnesutrymme, blir det exekveringsfel och programmet kraschar pga minneskonflikt.

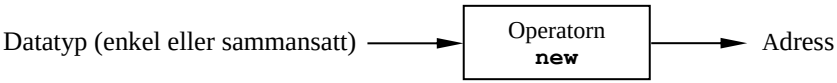
Ett annat lärorikt test är att i **Value** byta ut satsen ***pekarVar = 15.6;** mot:

```
*((double *) 0x12FF78) = 15.6;
```

Programmet kommer att fungera precis som förut, förutsatt att adressen som använts i denna sats är samma adress som **pekarVar**:s värde. Observera att sådana adresser blir olika när man kör programmet på annan dator eller vid olika tillfällen på samma dator. Det bästa är att ta adressen vid en aktuell körning från pekarvariabelns värde (se körresultatet ovan). Observera att den första asterisken är värdeoperatorn medan den andra tillhör den explicita typkonvertering som omvandlar den råa adressen till en adress-till-**double**. Testet visar att värdeoperatorn som vanligen skrivs framför en pekarvariabel, även fungerar när man skriver den framför en konstant adress. I så fall måste den konverteras till rätt datatyp och peka på ett allokerat minnesutrymme, i det här fallet variabeln **vanligVar** som även på det här sättet kan få värdet **15.6**.

10.4 Operatorn `new`

I alla pekarprogram hittills har vi använt både vanliga och pekarvariabler genom att tilldela de vanliga variablers adresser till pekarvariablerna. För att kunna göra det var vi dock tvungna att deklarera vanliga variabler. Nu går vi ett steg vidare: Vi eliminerar de vanliga variablerna och använder endast pekarvariabler. Men hur allokeras minne om vi inte deklarerar vanliga variabler? Svaret är operatorn **new** som vi ska behandla i detta avsnitt. **new** tar emot en *datatyp* som parameter, allokerar minne av den storlek som datatypen föreskriver och returnerar det allokerade minnesutrymmets *adress*:



En *operator* fungerar som en funktion med returvärde. Den tar emot ett antal parametrar, gör något och returnerar ett värde. Enda skillnaden är syntaxen: operator använder inga parenteser till skillnad från funktion. **new** är en fördefinierad minnesallokeringsoperator i C++. Skillnaden till traditionell allokering med vanlig deklaration är att **new** allokerar minne under programmets körning – *at run time* medan vanlig deklaration gör det under kompileringen – *at compile time*. Vi har alltså att göra med *dynamisk minnesallokering*. Efterom operatorn **new**:s returvärde är en adress måste den tilldelas en pekarvariabel. Man vill ju kunna referera till data som lagras vid denna adress. Data som man inte kan referera till dvs inte kan komma åt, är ju meningslös. Därför ser den generella syntaxen för operatorn **new** ut så här:

```
datatyp *pekarvariabel = new datatyp;
```

Ex.: `int *pekInt = new int;`

Själva **new**:s syntax står till höger om tilldelningstecknet. Till vänster deklarareras en pekarvariabel som tar hand om **new**:s returvärde. Här ser man också att operatorn **new** inte behöver parenteser kring sin parameter datatyp, vilket skiljer en operator från en funktion med returvärde. Observera att **new int** allokerar minne för lagring av en **int** och returnerar en adress till en **int**. Därför måste den också tilldelas en variabel av typ pekare-till-**int**. Tilldelning till en pekare-till-annan datatyp ger kompileringsfel.

I tidigare program hänvisade vi till minnescellerna direkt med *variabelnamn*. Nu gör vi samma sak indirekt med deras *adresser*. Vilken metod som är ”direkt” och vilken ”indirekt” kan man ha olika åsikter om. När man vant sig vid att använda pekare kan man t.o.m. tycka att hanteringen av data via adresser är det naturliga sättet, vilket inte är någon dum idé med tanke på att variabelnamn ändå är en slags användarvänlig mjukvarulänk till minnescellens adress i hårdvaran. I alla fall är det fullt *möjligt* att eliminera vanliga variabler och jobba endast med pekare. Om det

också är *meningsfullt* är en annan fråga som måste avvägas från fall till fall. Ofta är en blandning en gångbar kompromiss.

Nästa program är ett exempel på användningen av operatorn **new**. Där finns inga vanliga variabler längre eftersom **new** allokerar minne och returnerar dess adress till en pekare som används som referens till minnescellen. När värdeoperatorn tillämpas på denna pekare ger den oss åtkomst till innehållet i minnescellen för att både läsa och skriva i den.

```
// New.cpp
// Adderar heltal tills man matar in 0
// Använder enbart pekarvariabler, inga vanliga variabler
// Operatorn new tar en datatyp, allokerar minne passande
// till den och returnerar adressen till minnesutrymmet
// Åtkomst till allokerat minne med hjälp av värdeoperatorn
#include <iostream>
using namespace std;

int main()
{
   int *pekInt = new int;      // Deklarerar och initierar pe-
                               // karvariabeln pekInt
   int *pekSum = new int(0);   // Nollsätter dessutom minnes-
                               // cellen som pekSum pekar på
   do
   {
      cout << "\nGe ett heltal (avsluta med 0): ";
      cin >> *pekInt;
      *pekSum += *pekInt;
   } while (*pekInt != 0);
   cout << "\nSumman av heltalen är:\t" << *pekSum << "\n\n";
}
```

Ett körexempel visar att programmets användare inte märker att koden endast använt pekare:

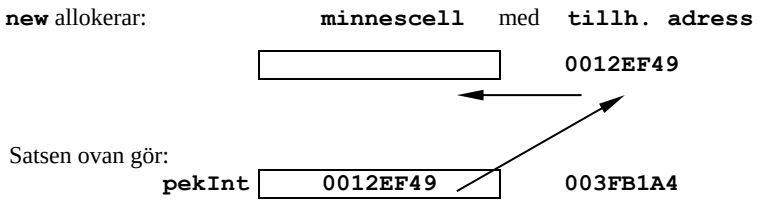
```
Ge ett heltal (avsluta med 0): 359

Ge ett heltal (avsluta med 0): 237

Ge ett heltal (avsluta med 0): 8

Ge ett heltal (avsluta med 0): -12

Ge ett heltal (avsluta med 0): 0

Summan av heltalen är:  592
```

Vad gör new?

Den första satsen i **main()**:

```
int *pekInt = new int;
```

allokerar minnesutrymme passande för lagring av ett **int**-värde och gör därmed samma sak som en vanlig deklarationssats – med skillnaden att den allokerade minnescellen inte får något namn. Istället returnerar **new** adressen. Vi tilldelar adressen i ovanstående sats till pekarvariabeln **pekInt** vilket gör att **pekInt** nu pekar på den av **new** allokerade minnescellen:



Samtidigt deklarerar satsen ovan **pekInt** som en variabel av typ pekare-till-**int** vilket även reserverar minne för **pekInt**. Detta är minnesekonomiskt inte någon fördel just i detta sammanhang, utan kommer att vara det först om pekaren pekar på större datamängder. Just nu har minnescellen inget deklarerat värde. Först i **do**-satsen läses in ett värde med:

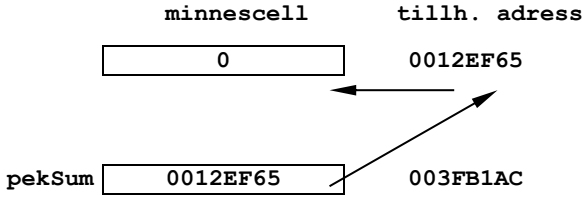
```
cin >> *pekInt;
```

som kommer att hamna i den tomma minnescellen ovan då pekaren **pekInt** pekar på den. Här betyder ***** värdeoperatorn som används för att komma åt minnescellen och skriva det inlästa värdet i den. Man kan säga att ***pekInt** ersätter det konventionella variabelnamnet.

Den andra satsen i **main()**:

```
int *pekSum = new int(0);
```

gör samma sak som den första plus att den samtidigt initierar minnescellen som **pekSum** pekar på med värdet **0**, så att minnesbilden efter satsen ovan ser ut så här:



Observera att tilldelningen av värdet **0** till minnescellen görs här på samma sätt som vi en gång kallade objektorienterad initiering – t.ex. **int sum(0);** istället för **int sum = 0;** – kan man även använda med **new**. Skillnaden är bara att det vanliga

variabelnamnet **sum** nu inte längre finns när **new** är med i bilden. Då skrivs initieringen **(0)** direkt efter datatypen. I objektorienterad programmering kommer denna initieringsteknik att få sin fulla förklaring och användas väldigt ofta med **new**.

Att minnescellen som **pekSum** pekar på, initieras till **0**, men inte minnescellen som **pekInt** pekar på, har att göra med programmets ändamål, men också med satsen:

```
*pekSum += *pekInt;
```

som är en kortform för och identisk med ***pekSum = *pekSum + *pekInt;** Ändamålet med programet **new** är att addera heltal tills man avslutar inmatningen med **0**. Det görs i en **do**-loop där de inmatade heltalen läses in och lagras i minnescellen som **pekInt** pekar på. Med satsen ovan adderas dessa tal med värden som finns i minnescellen som **pekSum** pekar på. Referensen ***pekSum** till denna minnescells värde i programmet förekommer även till höger om tilldelningstecknet i den kod som satsen ovan är en kortform för. Därför innehåller den ett odefinierat skräpvärde, när satsen utförs i allra första varvet av **do**-loopen. Då alla tal som matas in, ackumuleras (läggs samman) i denna minnescell, kommer de adderas till detta skräpvärde om vi inte nollsätter den i början. Detta kommer att resultera i ett felaktigt skräpvärde som summa.

Däremot kommer minnescellen som **pekInt** pekar på att initieras vid inmatningen. Därför behöver den inte initieras explicit i programmet även om det inte vore fel att göra det. I följande körexempel initieras ***pekInt** med värdet **359** som skriver över det befintliga skräpvärdet. **do**-loopens villkor ***pekInt != 0** gör att programmet kan avslutas med inmatning av **0** vilket gör villkoret falskt.

Programmet **new** syftade åt att bekanta oss med operatorn **new** och visade möjligheten att i C++ programmera enbart med pekare. Det är fullt möjligt att skriva en pekarversion av alla program – en bra övning för nybörjare som vill lära sig pekare. Däremot visade **new** inte hur dynamisk minnesallokering *egentligen* går till trots att minnet allokerades dynamiskt med **new**. Intressant och praktiskt relevant blir dynamisk minnesallokering först när man frigör eller utökar ett redan allokerat minne senare i programmet.

10.5 Pekare och array

När man fortsätter att programmera med pekare i C++ upptäcker man snart att det finns ett släktskap mellan pekare och array. Arrayen kan beskrivas som en användarvänlig variant av pekare.

Detta är också anledningen, varför andra programmeringsspråk har ”avskaffat” pekare. I C#, Java, Python, ... t.ex. har man utnyttjat detta släktskap för att bli av med pekare. Medan man i dessa språk arbetar med arrays, objekt och referenser görs förstås all minnesallokering med adresser. Ett datorprogram kan inte fungera utan adressering, oavsett programmeringsspråk. Att dessa språk fungerar utan pekare beror på att array är en variant av pekare som kan ersätta den. Man struntar då i vissa möjligheter av minnesmanipulation. I C++ har man som ett arv från C kvar möjligheten att styra adresshanteringen från koden. Hur släktskapet mellan pekare och array ser ut och vilka konsekvenser det har, vill vi studera i detta avsnitt.

Vi börjar med att skriva om programmet **ArrayDef** (sid 91) till en pekarversion:

```
// PointArray.cpp
// Pekarversion av programmet ArrayDef (sid 91)
// Skapar med new en array av int, tilldelar den elementvis
// med värdeoperatorn och kopierar värdena till en annan
// array. Pekarstegning ersätter indexering.
#include <iostream>
using namespace std;
int main()
{
   int *no = new int[4];        // Deklarerar och tilldelar
                                // pekarvariabeln no. new
                                // allokerar minne och no pe-
                                // kar på 1:a elementet i det.
   *no     = 64;                // Samma som *(no+0) = 64
                                // Elementvis tilldelning
   *(no+1) = 86;                // Pekarstegning: pekarens
   *(no+2) = 34;                // position flyttas fram
   *(no+3) = -6;
   int *copy = new int[4];      // Ny pekare copy
   for (int i=0; i <= 3; i++)
       *(copy+i) = *(no+i);     // Elementvis tilldelning
   cout << '\n';
   for (int i = 0; i <= 3; i++)
       cout << "\tcopy:s " << i+1 << ":a element *(copy+"
            << i << ") = " << *(copy+i)
            << " med pekarposition " << i << "\n\n";
}
```

Efter tilldelningen av arrayen **no** skapas en andra array, även den med **new**, som en ny pekare **copy** pekar på. Värdena i **no** kopieras elementvis över till **copy** i en **for**-sats. Slutligen skrivs ut värdena i **copy** samt deras position, vilket visar att kopieringen har gått bra:

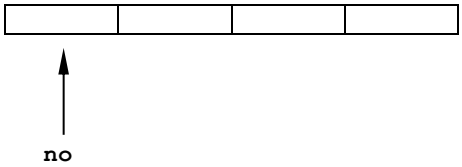
```
copy:s 1:a element *(copy+0) = 64 med pekarposition 0
copy:s 2:a element *(copy+1) = 86 med pekarposition 1
copy:s 3:a element *(copy+2) = 34 med pekarposition 2
copy:s 4:a element *(copy+3) = -6 med pekarposition 3
```

new skapar array

Den första satsen i **main()**:

```
int *no = new int[4];
```

deklarerar en pekare-till-**int** kallad **no**. Samtidigt allokerar **new** minne för en array av **int** med **4** element som **no** kommer att peka på, vilket i praktiken innebär att peka på det första elementet i arrayen. Följande minnesbild uppstår:



Dvs ett sammanhängande minnesområde bestående av **4** minnesceller där varje minnescell kan lagra ett **int**-värde. Till skillnad från den ursprungliga arrayversionen **ArrayDef**, saknar denna array namn eftersom den skapats med **new**. Arraynamnet har ersatts av pekaren **no** som vi kan jobba med istället. Att tilldela den *första* minnescellen ett värde, är inget problem: I den andra satsen av **main()**

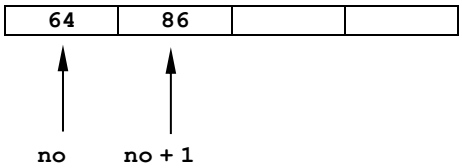
```
*no = 64;
```

gör värdeoperatorn detta: **no** pekar på arrayens första minnescell. Men hur kan vi tilldela den *andra* minnescellen ett värde? Eftersom arrayen saknar namn kan vi ju inte komma åt elementen med *index* vilket vi gjort hittills. Lösningen är:

```
*(no + 1) = 86;
```

Pekarstegning

Pekarens position flyttas fram *ett* steg in i arrayen:



Pekaren pekar nu på den andra minnescellen, ser minnesbilden ovan.

I koden åstadkommer man detta genom att skriva **no+1**. Additionsoperatorn betyder här inte matematisk addition, inte konkatenering utan pekarstegning, dvs flyttning av pekarens position ett steg framåt. Sedan tillämpas värdeoperatorn på den nya positionen: ***(no+1)** så att det refereras till värdet i den minnescell som **no+1** pekar på. Pekarstegning ersätter alltså indexering. Vi kan istället för index som vi använde hos arrays, använda pekaren, för att komma åt arrayens element.

Arraynotation vs. pekarnotation

Vad gäller åtkomsten till arrayelementen kan släktskapet mellan array och pekare bl.a. beskrivas generellt så här:

no[n] gör samma sak som ***(no + n)**

där **no** till vänster är en array och **n** arrayens index, medan **no** till höger är en pekare och **n** antalet steg vid pekarstegningen. Dessutom är ***** värdeoperatorn. Koderna ovan är ömsesidigt utbytbara. Till vänster har vi *arraynotation*, till höger *pekarnotation*. I ett och samma program kan arraynotation blandas med pekarnotation efter att allokering av minnesutrymme skett på ett korrekt sätt. För specialfallet **n = 0** följer av den generella regeln ovan:

```
no[0]      gör samma sak som    *(no + 0)
                                *(no)
                                * no
```

där arraynotationen visar det *första* arrayelementet. Men vad gäller pekarnotationen är ***(no + 0)** identiskt med ***(no)** och därmed med ***no**. Detta känner vi igen som värdet i den minnescell som **no** pekar på och som vi redan använt för att initiera det *första* arrayelementet: `*no = 64;`

Varför börjar arrayindex med 0 ?

Nu kan vi också förklara indexregeln för arrays (sid 90) enligt vilken numreringen av index i arrays alltid börjar med 0. Indexering av en array börjar med 0 därför att, när arrayen skapas vid deklationen, pekaren som pekar på det första elementet, inte stegats dvs antalet steg vid pekarstegningen är 0. Först när vi flyttar pekaren till nästa position så att den pekar på det andra elementet i arrayen, blir antalet steg vid pekarstegningen 1 och därmed blir också indexet 1. På samma sätt går det vidare.

Pekararitmetik

När vi pratar om *pekarstegning* och att *flytta pekarpositionen* menar vi egentligen det som kallas *pekararitmetik*. Med detta begrepp vill man anknyta till vanlig räkning och se på additionen i **no + 1** som vanlig addition av två hexadecimala tal. Vi sade att additionsoperatorn här betyder stegning av pekarens position vilket är en populär tolkning. I själva verket handlar det om att addera den hexadecimala adressen som lagras i pekarvariabeln **no** med 4. Orsaken till att pekarstegning med **1** steg innebär addition med 4, är att **no** är en pekare-till-**int** och **int** tar 4 bytes.

Hur många bytes **1** steg vid pekarstegning motsvarar beror alltså på datatypen som pekaren pekar på. På samma sätt hanteras indexet hos arrays. Mellan **no[0]** och **no[1]** ligger 4 bytes om **no** är en array av **int**. I arrays har man byggt in den användarvänliga tolkningen av pekarstegning som index. I hela detta resonemang är det underförstått att en array alltid är ett sammanhängande minnesområde och att adressering görs bytevis. Följande lilla test illustrerar resonemanget:

```
// PointArithm.cpp
// Skriver ut adresserna till elementen i en int-array.
// Omvandlar dem till decimala tal med explicit typkonverte-
// ring. Differensen är lika med minnesstorleken för en int
#include <iostream>
using namespace std;

int main()
{
   int *no = new int[4];
   cout << '\n';
   for (int i=0; i <= 3; i++)
    cout << "no:s " << i+1 << ":a element lagras vid adress "
            << no+i << " = " << (int) (no+i) << '\n';
                          // Omvandling till decimalt format
   cout << "\nAdressernas differens är "
        << (int) (no+1) - (int) no;
   cout << "\n";
}
```

Pekaren **no** pekar på **int**-arrayen skapad av **new** och därmed på det första elementet i den. **for**-satsen skriver ut adresserna **no**, **no+1**, **no+2** och **no+3** både i hexadecimalt och decimalt format. Omvandling till decimalt format sker i **for**-satsen:

```
(int) (no+i)
```

med explicit typkonvertering. Adresserna som är hexadecimala tal av typ **int *** dvs pekare-till-**int** konverteras till vanliga heltal av typ **int**. Omvandling till decimalt format visar redan att adressernas differens är 4 när vi kör:

```
no:s 1:a element lagras vid adress 00481C10 = 4725776
no:s 2:a element lagras vid adress 00481C14 = 4725780
no:s 3:a element lagras vid adress 00481C18 = 4725784
no:s 4:a element lagras vid adress 00481C1C = 4725788

Adressernas differens är 4
```

Men för att illustrera pekararitmetik beräknar vi denna differens genom att subtrahera de till decimalt format omvandlade första två adresserna från varandra:

```
(int) (no+1) - (int) no;
```

10.6 Stränghantering med pekare

Släktskapet mellan pekare och array som vi observerade i programmet **PointArray** (sid 288) blir ännu mer påtagligt när vi går över från pekare-till-**int** till pekare-till-**char**. Det beror på att strängar i C++ är teckenkedjor som avslutas med nolltecknet. Man kan låta en pekare peka på en **char** som då blir början av en sträng och sätta nolltecknet i slutet av strängen. Strängens längd spelar ingen roll. Detta gäller inte för pekare-till-**int**. Det som gör stränghanteringen med pekare mer intressant än med array, är möjligheten till dynamisk minnesallokering. Med pekare som verktyg för stränghantering blir man av med begränsningen som arrayens statiska minneshantering medför.

I följande program ska vi använda datatypen *pekare-till-**char*** för att låta programmet hitta initialerna i vårt för- och efternamn, vilket vi redan löst med array av **char**. Vi börjar även här med att lagra en sträng i en **char**-array. Men sedan skickas arrayens adress till en funktion som tar reda på samt skriver ut namnets första och andra initial. Funktionen tar emot arrayens adress via sin parameter av typ pekare-till-**char** och bearbetar strängen med denna pekare, vilket kommer att resultera i en enkel men samtidigt elegant kod: **while**-loopen som går igenom strängen behöver ingen extra indexvariabel.

```
// Initials.cpp
// Läser in för- och efternamn, skickar strängens pekare till
// funktionen initials() som hittar och skriver ut initia-
// lerna. Stränghantering med datatypen pekare-till-char
// Pekare som parameter i en funktion
#include <iostream>
using namespace std;

void initials(char *namn)     // Funktion med pekaren namn som
{                             // formell parameter
   cout << *namn;             // Ger första initialen
   while (*namn++ != ' ')     // Går igenom endast förnamnet
      if (*namn == ' ')       // Hittar namnets sista bokstav
         cout << *(namn + 1);  // Ger andra initialen
}

int main()
{
   char dittNamn[20];
   cout << "Ge ditt för- och efternamn: ";
   cin.getline(dittNamn, 20);
   cout << "\nHej " << dittNamn << ", dina initialer är: "
                                                 << " \n\n\t";
   initials(dittNamn);       // Funktionsanrop med arraynamnet
   cout << "\n";             // som aktuell parameter
}
```

Programmet **Initials** ger följande utskrift när jag matar in mitt namn:

```
Ge ditt för- och efternamn: Taifun Alishenas

Hej Taifun Alishenas, dina initialer är:

        TA
```

Första initialen **T** skrivs ut i funktionen **initials()** med satsen **cout << *namn;** då **namn** vid funktionsanropet fått sitt värde överfört från **dittNamn** som är adressen till den första bokstaven i strängen **Taifun Alishenas**. Tecknet som är lagrat vid denna adress dvs ***namn** är strängens första bokstav **T**.

Den andra initialen **A** skrivs ut med satsen **cout << *(namn+1);** som finns i **if**-satsen. I varje varv av loopen testar **if**-satsen om det aktuella tecknet ***namn**, är mellanslaget. Detta är endast fallet när sökningen har nått förnamnets sista bokstav. Då skrivs ut det tecken som kommer efter det aktuella tecknet dvs ***(namn+1)**. Men detta är efternamnets första bokstav dvs den andra initialen. Observera att **while**-satsen endast söker igenom förnamnet. Den använder ingen information om strängens längd. Vi behöver inte skicka arrayens storlek till funktionen.

Array som konstant pekare

Här ska vi titta på hur strängen hamnar i funktionen. Via funktionsanropet förstås:

```
initials(dittNamn);
```

Arrayen **dittNamn** är definierad i **main()**. Vi läser in den med **cin.getline()** för att få in hela namnet inkl. mellanslaget mellan för- och efternamn. Vid anropet överförs arrayen **dittNamn** till pekarvariabeln **namn** i funktionen **initials()**. Men *hur* sker denna parameteröverföring? Vad är det exakt som skickas? Är det hela namnet – inkl. mellanslaget – som skickas eller är det endast adressen? Om det är namnet, som är en array, hur kan den tas emot av pekaren **namn** i funktionen? Detta är bara möjligt om namnet är en adress. Slutsats:

> Ett arraynamn är en konstant pekare.

Med ”konstant pekare” menas ett fast adressvärde. Man kan alltid tilldela ett arraynamn till en pekarvariabel p.g.a. denna egenskap. En gång skapad kan en konstant pekare inte ändras i programmet. En pekarvariabel däremot kan initieras till ett adressvärde, ändra sedan sitt värde och anta ett nytt värde. Att vi hittills ändå pratat om array*variabler* syftar snarare åt arrayens *element*, inte på namnet. Arrayelementen är variabla och deras värden kan ändras. De lagras däremot alltid vid samma adress som i programmet representeras av arraynamnet.

while-loopen

Efter anropet av funktionen **initials()** och parameteröverföringen pekar nu pekaren **namn** på den inlästa sträng som innehåller både för- och efternamn. Att hitta den första initialen är inte svårt: Satsen **cout << *namn;** skriver ut den eftersom ***namn** är värdet som pekaren **namn** pekar på. Eftersom vi i det här läget inte ändrat pekarpositionen pekar **namn** på den *första* bokstaven i den inlästa strängen som är den första initialen. Den andra initialen hittas i loopen:

```
while (*namn++ != ' ')
   if (*namn == ' ')
      cout << *(namn + 1);
```

När vi börjar står pekarpositionen på strängens första bokstav. Loopen stoppas när sökningen stöter på ett mellanslag – tecknet mellan för- och efternamnet. Därför förutsätter programmet **Initials** att användaren matar in sitt för- och efternamn skilda med endast *ett* mellanslag – det vanliga sättet alltså att ange sitt namn. Så länge det aktuella tecknet ***namn** inte är mellanslag fortsätter loopen vilket framgår av loopens villkor ***namn++ != ' '** . Men vad gör ökningsoperatorn **++** direkt efter ***namn**? Utan den vore det som sades, mer begripligt. Jo, ökningsoperatorn **++** har inget med villkoret att göra utan den ska se till att pekarpositionen flyttas fram med ett steg i varje varv av loopen. På så sätt går man igenom strängen. Observera att detta är pekarstegning (sid 290). Dvs ökningsoperatorn **++** opererar på pekaren, därför att den står direkt efter **namn**, medan före pekaren står värdeoperatorn ***** för att hämta värdet (tecknet) som **namn** pekar på. Så uppstår koden ***namn++** där två olika operatorer tillämpas på **namn**. Och det roliga är: De stör inte varandra. Var och en utför sitt jobb oberoende av den andra. Men *när* exakt sker stegningen av pekarpositionen? Är det före, efter eller i **while**-satsen? Är det före eller efter loopens **if**-sats? Eftersom vi använt ökningsoperatorns postfixvariant kan det inte vara för. Faktum är att pekarstegningen görs *efter att* **while**-*satsens villkor testats*, vilket görs i början av varje varv strax innan **if**-satsen.

En enklare variant

Man får en enklare variant när man flyttar pekarstegningen från loopens villkor till loopens kropp, närmare bestämt till det ovan beskrivna stället i koden, dvs om man byter ut programmets **while**-sats med:

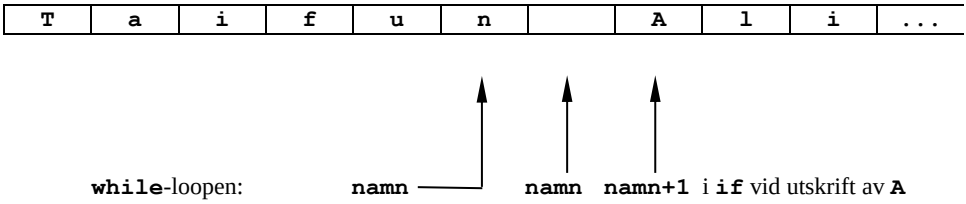
```
while (*namn != ' ')
{
   namn++;
   if (*namn == ' ')
      cout << *(namn+1);
}
```

Låt oss i tankarna frysa programkörningen till det kritiska ögonblick då den andra initialen **A** skrivs ut. Vid denna utskrift står pekaren **namn** på en position som pekar på mellanslaget. Då är för första gången **if**-satsens villkor ***namn** == **' '** uppfyllt.

Därför utförs **cout**-satsen som står i den. Man kan undra hur det står till med den övergripande **while**-satsens villkor vid denna tidpunkt. Det är ju just motsatsen till **if**-satsens villkor. Faktum är att **while**-satsens villkor inte är uppfyllt vid denna tidpunkt. Men det gör inget eftersom det var uppfyllt när det testades och vi kom in i loopens sista varv. Anledningen till det är att mellan **while**-villkorets test och **if**-villkoret står pekarstegningen **namn++** oavsett om den är inbakad i **while**-villkoret eller separat i början av loopen. Mellan **while**- och **if**-villkoret flyttas pekaren ett steg. Dvs **while** testas när pekaren pekar på bokstaven **n** och är skilt från mellanslag. **if** testas när pekaren pekar på mellanslag. Efter **if** är **while**-satsens villkor falskt och loopen avbryts. Det är loopens sista varv som skriver ut den andra initialen. Det är därför **while**-satsen endast går igenom förnamnet. Vi behöver inte bry oss om när strängen tar slut. Nolltecknet behöver därför inte användas som avslutningsvillkor i **Initials**.

Arrayens minnesbild

I **while**-satsens sista varv har vi följande bild av arrayen. Då pekar nämligen pekaren **namn** på **n**. Men sedan stegas pekaren med 1, så att den pekar på mellanslaget när **if**-satsen testas och **A** skrivs ut som pekaren **namn+1** pekar på:



Array vs. pekare

Vad händer om man hela funktionen **initials()** skrivs om till arraynotation?

```
void initials(char namn[])
{
   int i=0;
   cout << namn[0];
   while (namn[i++] != ' ')
      if (namn[i] == ' ')
         cout << namn[i+1];
}
```

Vid denna omskrivning kan resten av programmet **Initials** dvs koden i **main()** bibehållas oförändrad och kommer att fungera utan problem – ett resultat av modularisering. Jämför man båda varianter kan man konstatera att arrayvarianten använder en extra indexvariabel **i** som inte behövs i pekarvarianten. Annars är det i princip smaksak att favorisera den ena eller den andra. Arrayvarianten har fördelen av bättre läslighet. Pekarvarianten kan tänkas föredras p.g.a. sin kompakthet och elegans av folk som har lite längre erfarenhet av programmering.