EDAF30 – Programmering i C++

5. Pekare och arrayer. Algoritmer.

Sven Gestegård Robertz
Datavetenskap, LTH

2017

Innehåll

Datatyper
Pekare, Arrayer och Referenser

- Pekare (heltalstyp, aritmetik, till skillnad från i Java)
- Referenser (konstanta och kan inte vara "`null`")[1]
- Arrayer (lågnivå sammanhängande sekvens av objekt)

[1] Egentligen inte en typ, utan ett *alias* till ett objekt

Pekare

Påminner om referenser i Java, men

- en pekare är *minnesadressen till ett objekt*
- en pekare *är själv ett objekt* (till skillnad från en referens)
  - kan tilldelas och kopieras
  - har en adress
  - måste inte initialiseras när den definieras men får då ett *odefinierat värde* , på samma sätt som andra variabler
- fyra möjliga tillstånd
  1. pekar på ett objekt
  2. pekar på adressen omedelbart efter ett objekt
  3. pekar på ingenting: `nullptr`. Före C++11: `NULL`
  4. är ogiltig (allt annat än ovanstående)
- kan användas som ett heltalsvärde
  - aritmetik, tilldelning, etc.

Var väldigt försiktiga!

Pekare
Syntax, operatorerna ∗ och &

- I en *deklaration*:
  - ∗: "pekare till"
    `int *p;` : p är *en pekare till en int*
    `void swap(int*, int*);` : *funktion som tar två pekare*
  - &: "referens till"
    `int &r;` : r är *en referens till en int*
- I ett *uttryck*:
  - prefix ∗: "innehållet i"
    `*p = 17;` *objektet som p pekar på* tilldelas värdet 17
  - prefix &: "adressen till", "pekare till" (*address-of*)

```
int x = 17;
int y = 42;

swap(&x, &y);
```

Anropa `swap()` med *pekare till* x *och* y

Pekare
Syntax

Deklaration `T*`

```
T* ptr;       // en pekare till T
```

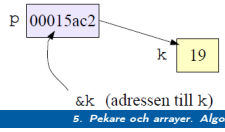
Referensoperatorn `&`, *address of*

Ger adressen till ett objekt

Dereferensoperatorn ∗

Ger värdet som pekaren pekar på, dvs det vars adress ligger lagrad i pekarvariabeln

```
int k;
int* p;
p = &k;
*p = 19;
```



Pekare
Var tydlig med deklarationer

Använd regeln "en deklaration per rad"

```
int* a;     // pekare till int
int *b;     // pekare till int
int  c;     // int-variabel

int* d, e;  // d är pekare till int, e är int-variabel
int *f, *g; // f och g är pekare till int
```


Pekare

Sammanfattning

```
typ *p;    // p får typen  "pekare till typ"

p = &v;    // p tilldelas adressen till v: "p pekar på v"

*p = 12;   // Objektet som p pekar på tilldelas värdet 12

p1 = p2;   // p1 pekar på samma objekt som p2  ‘‘pekar-tilldelning’’

*p1 = *p2; // Objektet som p1 pekar på tilldelas
           // värdet som p2 pekar på           ‘‘värde-tilldelning’’
```

Pekare är objekt

Klasser
Åtkomst av klassmedlemmar

```
Struct Point{
    int x;
    int y;
}

Point p;

Point& rp;

Point* pp;

...

int i = p.x;     // åtkomst via namn (på variabel)

int j = rp.x;    // åtkomst via referens (alias för namn)

int k = pp->x;   // åtkomst via pekare
```


Access av medlemmar via pekare
Operatorn ->

Givet en pekare p, så kan vi uttrycka
"Medlemmen x i objektet som p pekar på" på två sätt:

- `p->x`
- `(*p).x`

Referenser

Referenser liknar pekare, men

- En referens är *ett alias till* en variabel
  - kan inte ändras (fås att peka på en annan variabel)
  - måste initieras
  - är inget objekt (har ingen address i minnet)
  - För dereferens används inte operatorn *
    - Att använda en referens *är* att använda objektet som refereras.

Använd referenser om du inte måste använda pekare.

- T ex om en variabel får ha värdet `nullptr` (*"inget objekt"*)
- eller om man behöver kunna ändra ("peka om") pekaren
- Mer om detta senare.

Pekare och referenser
Pekaranrop

I vissa fall används *pekare* istället för *referenser* för att göra "referensanrop":

Exempel: Byta plats på två heltalsvärden

```
void swap2(int* a, int* b)
{
  if(a != nullptr && b != nullptr) {
    int tmp=*a;
    *a = *b;
    *b = tmp;
  }
}  ...  och användning:       int x, y;
                               ...
                               swap2(&x, &y);
```

Notera:

- en pekare kan vara `nullptr` eller oinitierad
- att dereferera sådan pekare ger *undefined behaviour*

Pekare och referenser

Pekarversion och referensversion av swap

```
// Referensversionen
void swap(int& a, int& b)
{

   int tmp = a;
   a = b;
   b = tmp;
}
```

```
// Pekarversionen
void swap(int* pa, int* pb)
{
  if(pa != nullptr && pb != nullptr) {
    int tmp = *pa;
    *pa = *pb;
    *pb = tmp;
  }
}
```

```
int m=3, n=4;
swap(m,n);    Referensversionen används

swap(&m,&n); Pekarversionen används
```

NB! *Värdeanrop*: *adressen* kopieras

Arrayer ("C-arrayer", "*built-in arrays*")

- En sekvens av värden av samma typ
- Likt Java för primitiva typer
  - men *inget skyddsnät* – Skillnad från Java
  - en array vet inte hur stor den är – programmerarens ansvar
- *Kan inehålla element av godtycklig typ*
  - Skillnad från Java, som bara kan ha *referenser till objekt* som element
- Deklareras `T a[storlek];` (Skillnad från Java)
  - Storleken måste vara *en konstant*. (Skillnad från C99)

Arrayer
Representation i minnet

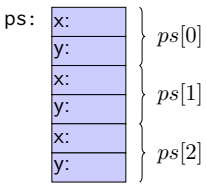
Elementen i en array kan vara av godtycklig typ

- Java: endast primitiv typ eller referens
- C++: objekt eller pekare

Exempel: array av `Point`

```
struct Point{
    int x;
    int y;
};

Point ps[3];
```



Arrayer ("C-arrayer", "*built-in arrays*")

Deklaration utan initiering

```
int x[7];   // innehåller odefinierade värden
```

Deklaration och initiering av `int`-array

```
int a[7] {0, 1, 2, 3, 4, 5, 6};
int b[20] {0, 1, 2, 3}; // resten av elementen initieras till 0
```

Utelämnande av längden

```
int b[] {1, 1, 0, 1, 0, 0, 1};
```

Initiering med = som i C

```
int a[7] = {0, 1, 2, 3, 4, 5, 6};
int b[]  = {1, 1, 0, 1, 0, 0, 1};
```


C-arrayer
tilldelning, djup kopiering

```
int a[7] = {0, 1, 2, 3, 4, 5, 6};
int b[7] = {1, 1, 0, 1, 0, 0, 1};
```

Ej tillåtna tilldelningar

```
b = a;                      invalid array assignment
b = {1, 1, 0, 1, 0, 0, 1}; assigning from an initializer list
```

Korrekt men omständligt: Kopiera elementvis

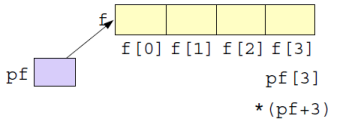
```
for (int i=0; i != 7; ++i){
  b[i]=a[i];
}
```


Pekare och arrayer

Arrayer kan adresseras m.h.a. pekare

```
float f[4];         // 4 st float
float* pf;          // pekare till float

pf = f;             // samma som pf = &f[0]
float x = *(pf+3); // Alt. x = pf[3];
x = pf[3];          // Alt. x = *(pf+3);
```



Pekare och arrayer
Vad betyder array-indexering egentligen?

Uttrycket `a[b]` är ekvivalent med `*(a + b)` (och därmed med `b[a]`)

Definition

För en pekare, `T* p`, och ett heltal `i`, så definieras `p + i` som

`p + i * sizeof(T)`

Exempel: förvirrande kod

```
int a[] {1,4,5,7,9};

cout << a[2] << " == "<< 2[a] << endl;
  5 == 5
```


Pekare och arrayer
Funktionsanrop

Funktion för nollställning av heltalsarrayer

```
void zero(int* x, size_t n) {
   for (int* p=x; p != x+n; ++p)
      *p = 0;
}
```

```
...
int a[5];

zero(a,5);
```

- *Namnet på en array* i ett uttryck tolkas som "pekare till första elementet": *array decay*
- a ⇔ &a[0]

Indexering av arrayer är oftast tydligare

```
void zero(int x[], size_t n) {
  for (size_t i=0; i != n; ++i)
    x[i] = 0;
}
```

- I funktionsparametrar är `T a[]` *ekvivalent* med `T* a`. (Syntaktiskt socker)
- `T*` används oftast

Multidimensional arrays

Flerdimensionella arrayer

- Finns (egentligen) inte i C++
  - utan är arrayer av arrayer
  - Ser ut som i Java
- Java: array av *referenser till arrayer*
- C++: två alternativ
  - Array av arrayer
  - Array av *pekare (till första elementet i en array)*

Tabeller, matriser

Initiering av tabeller med initierarlista

3 rader, 4 kolumner

```
int a[3][4] = {
  {0, 1, 2, 3} ,   /*  initierare för rad 0 */
  {4, 5, 6, 7} ,   /*  initierare för rad 1 */
  {8, 9, 10, 11}   /*  initierare för rad 2 */
};
```

I stället för kapslade initierarlistor kan man skriva:

```
int a[3][4] = {0,1,2,3,4,5,6,7,8,9,10,11};
```

- Flerdimensionella arrayer lagras som en array i minnet.
- Dimensionen *närmast namnet* är storleken på arrayen
- Resterande dimensioner tillhör element-typen

Tabeller, matriser
Flerdimensionemma arrayer i minnet

En deklaration `T array[4]` lagras i minnet med fyra element av typen `T`, efter varandra: | `T` | `T` | `T` | `T` |

För deklarationen **`int`** `a[3][4]` läggs 3 st **`int`**`[4]` ut efter varandra:
| **`int`**`[4]` | **`int`**`[4]` | **`int`**`[4]` |

Varje **`int`**`[4]` har strukturen

| **`int`** | **`int`** | **`int`** | **`int`** |

Alltså läggs **`int`**`[3][4]` ut som

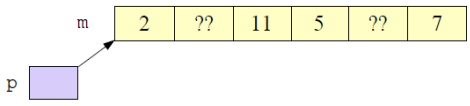
| **`int`** | **`int`** | **`int`** | **`int`** | **`int`** | **`int`** | **`int`** | **`int`** | **`int`** | **`int`** | **`int`** | **`int`** |

Tabeller, matriser

Flerdimensionella arrayer

```
int m[2][3]; // En 2x3-matris

m[1][0] = 5;
int* p;
p = m;       // Fungerar inte!
p = &m[0][0];
*p = 2;
p[2] = 11;
int* q=m[1];
q[2] = 7;
```



Tabeller, matriser

Parametrar av typ flerdimensionella arrayer

```
// Ett sätt att deklarera parametern
void printmatr(int (*a)[4], int n);

// Ett annat. tydligare?
void printmatr(int a[][4], int n) {
{
   for (int i=0;i!=n;++i) {
      for (int j=0;j!=4;++j) {
         cout << a[i][j] << " ";
      }
      cout << endl;
   }
}
```


Tabeller, matriser

Initiering och anrop

```
int a[3][4] {1,2,3,4,5,6,7,8,9,10,11,12};
int b[3][4] {{1,2,3,4},{5,6,7,8},{9,10,11,12}};

printmatr(a,3);
cout << "-----------------" << endl;
printmatr(b,3);
```

```
       1        2        3        4
       5        6        7        8
       9       10       11       12
-----------------
       1        2        3        4
       5        6        7        8
       9       10       11       12
```


Algoritmer

Tillgång till standardalgoritmer i C++ fås med direktivet

```
#include <algorithm>
```

Numeriska algoritmer:

```
#include <numeric>
```

Appendix A.2 i boken ger en översikt.

Algoritmer

Huvudkategorier av algoritmer

1. Söka, räkna
2. Jämföra, genomlöpa
3. Generera nya data
4. Kopiera och flytta element
5. Ändra och ta bort element
6. Sortera
7. Operationer på sorterade datasamlingar
8. Operationer på mängder
9. Numeriska algoritmer

Algoritmer
Exempel: `find`

```
template <class InputIterator, class T>
InputIterator find (InputIterator first, InputIterator last,
                    const T& val);
```

Exempel:

```
vector<std::string> s{"Kalle", "Pelle", "Lisa", "Kim"};

auto it = std::find(s.begin(), s.end(), "Pelle");

if(it != s.end())
    cout << "Hittade " << *it << endl;
else
    cout << "Mislyckades"<< endl;
```

```
Hittade Pelle
```


Algoritmer
Exempel: `find_if`

```
template <class InputIterator, class UnaryPredicate>
InputIterator find_if (InputIterator first, InputIterator last,
                       UnaryPredicate pred);
```

Exempel:

```
bool is_odd(int i) { return i % 2 == 1; }

void test_find_if()
{
    vector<int> v{2,4,6,5,3};

    auto it = std::find_if(v.begin(), v.end(), is_odd);

    if(it != v.end())
      cout << "Hittade " << *it << endl;
    else
      cout << "Mislyckades"<< endl;
}
```

```
Hittade 5
```

Funktionspekare

Algoritmer
count och count_if

Räkna element, ur en datastruktur, som uppfyller något vilkor

- std::count(first, last, value)
  - element lika med value
- std::count_if(first, last, predicate)
  - element för vilka predicate ger true

Algoritmer
Exempel: copy och copy_if

```
template <class InputIterator, class OutputIterator>
OutputIterator copy (InputIterator first, InputIterator last,
                     OutputIterator result);
```

Exempel:

```
vector<int> a(8,1);

print_seq(a);          length = 8: [1][1][1][1][1][1][1][1]

vector<int> b{5,4,3,2};

std::copy(b.begin(), b.end(), a.begin()+2);
print_seq(a);          length = 8: [1][1][5][4][3][2][1][1]
```

copy_if *analogt med tidigare*

Algoritmer
Insättningsiteratorer <iterator>

Exempel:

```
vector<int> v{1, 2, 3, 4};

vector<int> e;
std::copy(v.begin(), v.end(), std::back_inserter(e));
print_seq(e);          length = 4: [1][2][3][4]

deque<int> e2;
std::copy(v.begin(), v.end(), std::front_inserter(e2));
print_seq(e2);         length = 4: [4][3][2][1]

std::copy(v.begin(), v.end(), std::inserter(e2, e2.end()));
print_seq(e2);         length = 8: [4][3][2][1][1][2][3][4]

std::vector<int> w;
std::copy_if(v.begin(), v.end(), std::back_inserter(w), is_odd);
print_seq(w);          length = 2: [1][3]
```


transform och funktionsobjekt

Iterera över en collection och applicera en funktion på varje element (*jfr funktionen "map" i funktionella språk*)

```
template < class InputIt, class OutputIt, class UnaryOperation >
OutputIt transform( InputIt first, InputIt last, OutputIt d_first,
                    UnaryOperation unary_op );

template < class InputIt1, class InputIt2, class OutputIt,
           class BinaryOperation >
OutputIt transform( InputIt1 first1, InputIt1 last1, InputIt2 first2,
                    OutputIt d_first, BinaryOperation binary_op );
```

Funktionsobjekt är objekt som kan anropas som funktioner.

- funktionspekare
- funktionsobjekt (*"functor"*)

Algoritmen transform kan hantera både funktionspekare och funktionsobjekt.

Funktionsobjekt och transform

Exempel med funktionspekare

```
int square(int x) {
    return x*x;
}

vector<int> v{1, 2, 3, 5, 8};
vector<int> w; // w är tom!

transform(v.begin(), v.end(), inserter(w, w.begin()), square);

// w = {1, 4, 9, 25, 64}
```


Funktionsobjekt

Ett funktionsobjekt är ett objekt från en klass som har en **operator()**

Föregående exempel med funktionsobjekt

```
struct {
    int operator() (int x) const {
        return x*x;
    }
} sq;

vector<int> v{1, 2, 3, 5, 8};
vector<int> ww;  // ww är också tom!

transform(v.begin(), v.end(), inserter(ww, ww.begin()), sq);

// ww = {1, 4, 9, 25, 64}
```

Anonym struct – *typen* har inget namn, bara *objektet*.

Funktionsobjekt
Fördefinierade funktionsobjekt: <functional>

Funktioner:

```
plus, minus, multiplies, divides, modulus, negate,
equal_to, not_equal_to, greater, less, greater_equal,
less_equal, logical_and, logical_or, logical_not
```

Fördefinierat funktionsobjekt skapas med

```
operation<typ>()
```

t ex

```
    auto f = std::plus<int>();
```


Funktionsobjekt
Exempel: std::plus ur <functional>

transform med binär funktion

```
vector<int> v1{1,2,3,4,5};
vector<int> v2(10,10);

vector<int> res2;
auto it = std::inserter(res2, res2.begin());
auto f = std::plus<int>();
std::transform(v1.begin(), v1.end(), v2.begin(), it, f);

print_seq(res2);
```

```
length = 5: [11][12][13][14][15]
```

Exempel med accumulate <numeric>

```
auto mul = std::multiplies<int>();
int prod = std::accumulate(v1.begin(), v1.end(), 1, mul);

cout << "product(v1) = " << prod << endl;
```

```
product(v1) = 120
```


Iteratorer

Kategorier av iteratorer:

- Input Iterator (++ == !=) (defererens som *rvalue*: *a, a->)
- Output Iterator (++ == !=) (defererens som *lvalue*: *a=t)
- Forward Iterator (Input- och Output Iterator, återanvändning)
- Bidirectional Iterator (som Forward Iterator, samt --)
- Random-access Iterator (+=, -=, a[n], <, <=, >, >=)

Olika iteratorer för en containertyp (con symboliserar någon av containertyperna vektor, deque eller list med elementtypen T)

con<*T*>::iterator	löper framåt
con<*T*>::const_iterator	löper framåt, endast för avläsning
con<*T*>::reverse_iterator	löper bakåt
con<*T*>::const_reverse_iterator	löper bakåt, endast för avläsning

istream_iterator<T>

istream_iterator<T> : konstruktorer

```
istream_iterator();  // ger en  end-of-stream istream iterator
istream_iterator (istream_type& s);
```

```
#include <iterator>

stringstream ss{"1 2 12 123 1234\n17\n\t42"};

istream_iterator<int> iit{ss};
istream_iterator<int> iit_end;

while(iit != iit_end) {
    cout << *iit++ << endl;
}
```

```
1
2
12
123
1234
17
42
```


istream_iterator<T>

Användning för att initiera vector<int>:

```
stringstream ss{"1 2 12 123 1234\n17\n\r42"};

istream_iterator<double> iit{ss};
istream_iterator<double> iit_end;

vector<int> v{iit, iit_end};

for(auto a : v) {
    cout << a << " ";
}
cout << endl;
```

```
1 2 12 123 1234 17 42
```


istream_iterator
Felhantering

```
stringstream ss2{"1 17 kalle 2 nisse 3 pelle\n"};
istream_iterator<int> iit2{ss2};
while(!ss2.eof()) {
  while(iit2 != iit_end) { cout << *iit2++ << endl; }
  if(ss2.fail()){
    ss2.clear();
    string s;
    ss2 >> s;
    cout << "ss2: not an int: " << s << endl;
    iit2 = istream_iterator<int>(ss2); // create new iterator
  }
}
cout << boolalpha << "ss2.eof(): " << ss2.eof() << endl;
```

```
1
17
ss2: not an int: kalle
2
ss2: not an int: nisse
3
ss2: not an int: pelle
ss2.eof(): true
```

- vid fel sätts fail-biten för strömmen
- iteratorn sätts till end
- om man ändrar strömmen måste man skapa en ny iterator

Iteratorers giltighet

Generellt, om man ändrar strukturen en iterator refererar in i *blir iteratorn ogiltig*. Exempel:

- insättning
  - sekvenser
    - `vector`, deque* : alla iteratorer blir ogitiga
    - `list` : iteratorer påverkas inte
  - associativa containers (`set`, `map`)
    - iteratorer påverkas inte
- borttagning
  - sekvenser
    - `vector` : iteratorer *efter* de borttagna elementen blir ogitiga
    - deque : alla iteratorer blir ogitiga (i princip*)
    - `list` : iteratorer till de borttagna elementen blir ogiltiga
  - associativa containers (`set`, `map`)
    - iteratorer påverkas inte
- storleksförändring (`resize`): som insättning/borttagning

`ostream_iterator` och algoritmen `copy`

`ostream_iterator`

```
ostream_iterator (ostream_type& s);
ostream_iterator (ostream_type& s, const char_type* delimiter);
```

```
stringstream ss{"1 2 12 123 1234\n17\n\r42"};

istream_iterator<double> iit{ss};
istream_iterator<double> iit_end;

cout << fixed << setprecision(2);
ostream_iterator<double> oit{cout, " <-> "};

std::copy(iit, iit_end, oit);
```

```
1.00 <-> 2.00 <-> 12.00 <-> 123.00 <-> 1234.00 <-> 17.00 <-> 42.00 <->
```


Läsanvisningar

Referenser till relaterade avsnitt i Lippman

Pekare och Referenser 2.3

Arrayer och pekare 3.5

Flerdimensionella arrayer 3.6

Algoritmer 10 – 10.3.1, 10.5

Iteratorer 10.4

Funktions-objekt 14.8

Nästa föreläsning

Resurshantering

Referenser till relaterade avsnitt i Lippman

Dynamiskt minne och smarta pekare 12.1

Dynamiskt allokerade arrayer 12.2.1

Klasser, resurshantering 13.1, 13.2