EDAF30 – Programmering i C++

4. Standard-containers. Pekare och arrayer

Sven Gestegård Robertz
Datavetenskap, LTH

2016

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

Deklaration `T*`

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

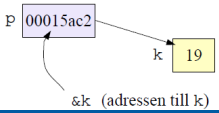
Referensoperatorn `&`, *address of*

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
Syntax, operatorer

- 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 `&`: "addressen till", "pekare till" (*address-of*)

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

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

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

Pekare
Var tydlig med deklarationer

Använd tumregeln: "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;    // Det som p pekar på tilldelas värdet 12

p1 = p2;    // p1 pekar på samma som p2          ''pekar-tilldelning''

*p1 = *p2; // Det som p1 pekar på tilldelas
           // värdet som p2 pekar på            ''värde-tilldelning''
```

Pekare är objekt

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*

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

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

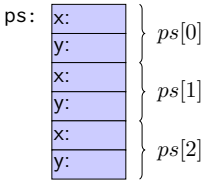
Elementen i en array kan vara av godtycklig typ

- 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

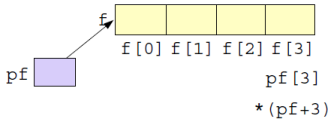
```
for (int i=0; i != 7; ++i){
  b[i]=a[i];
}
```


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;

cout << 4[a] << " == "<< a[4] << endl;
```

```
5 == 5
9 == 9
```


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.

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

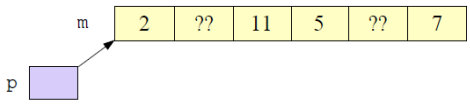
`| int | int | int | int | int | int | int | int | int | int | int | int |`

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)[3], int n);

// Ett annat. tydligare?
void printmatr(int a[][3], int n)
{
   for (int i=0;i!=n;++i) {
      for (int j=0;j!=3;++j) {
         cout << a[i][j] << " ";
      }
      cout << endl;
   }
}
```


Tabeller, matriser

Initiering och anrop

```
void printmatr(int a[][3], int n);

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

printmatr(a,4);
cout << "----------------" << endl;
printmatr(b,4);
```

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

Containerklasser – Klassificering

Sekvenser (homogena)

- `vector<T>`
- `deque<T>`
- `list<T>`

Associativa containers (finns även *unordered*)

- `map<K,V>`, `multimap<K,V>`
- `set<T>`, `multiset<T>`

Heterogena sekvenser (inte "containers")

- `tuple<T1, T2, ...>`
- `pair<T1,T2>`

Klasserna `vector` och `deque`

Operationer i klassen `vector`

```
v.clear(), v.size(), v.empty()
v.push_back(), v.pop_back()
v.front(), v.back(), v.at(i), v[i]
v.assign(), v.insert(), v.emplace()
v.resize(), v.reserve()
```

Ytterligare operationer i klassen `deque`

```
d.push_front(), d.pop_front()
```


Klasserna `vector` och `deque`
Konstruktorer och funktionen `assign`

Konstruktor och `assign` finns i tre varianter:

- *fill*: n stycken element med samma värde

```
void assign (size_type n, const value_type& val);
```

- *initializer list*

```
void assign (initializer_list<value_type> il);
```

- *range*: kopierar från `first` till `last` (exkl `last`, $[first, last)$)

```
template <class InputIterator>
void assign (InputIterator first, InputIterator last);
```


Klasserna `vector` och `deque`
Medlemsfunktionen `assign`, exempel

```
vector<int> v;
int a[]{0,1,2,3,4,5,6,7,8,9};

v.assign(3,1);
print_seq(v);          length = 3: [1][1][1]

v.assign({11,13,15,17});
print_seq(v);          length = 4: [11][13][15][17]

v.assign(a, a+5);
print_seq(v);          length = 5: [0][1][2][3][4]

std::deque<int> d;
d.assign(v.begin(), v.end());
print_seq(d);          length = 5: [0][1][2][3][4]
```

Exempel på iteratorer

Klasserna `vector` och `deque`
Medlemsfunktioner `push` och `pop`

`push` lägger till ett element, storleken ökar

`pop` tar bort ett element, storleken minskar

`*_back` opererar på slutet, finns i båda

```
void push_back (const value_type& val);    //copy
void pop_back();
```

bara i deque: `*_front`

```
void push_front (const value_type& val);   //copy
void pop_front();
```


Iteratorer

Iterator

"Pekarliknande" variabel som används för att genomlöpa en struktur (datasamling)

Exempel

```
vector<double> v(4);

vector<double>::iterator it;
for (it=v.begin(); it != v.end(); ++it)
    *it = 0;

for (auto it=v.begin(); it != v.end(); ++it) // med auto (C++11)
    *it = 0;
for (double &e : v)  // Ekvivalent i C++11
    e = 0;
```

Iteratorer

Funktioner som returnerar iteratorer och som finns i alla containerklasser + klassen `string`

`begin()`	ger en iterator som pekar på det första elementet
`end()`	ger en iterator som pekar på ett tänkt element *efter* det sista elementet
`rbegin()`	ger en reverserad iterator som pekar på det sista elementet
`rend()`	ger en reverserad iterator som pekar på ett tänkt element *före* det första elementet

samt:

```
cbegin()    rcbegin()
cend()      rcend()
```


Iteratorer

Operationer på en sekvens med iteratorer som parametrar (iteratorintervallet `[i,j)` betecknar intervallet från och med `i` till och med positionen före `j`)

sekv<typ> s(i,j);	konstruktor, s initeras med `[i,j)`
s.assign(i,j);	s = elementen i intervallet `[i,j)`
...	
s.insert(p,e);	sätter in värdet `e` i positionen (iterator) `p`
s.insert(p,n,e);	sätter in `n` stycken `e` i positionen `p`
s.insert(p,i,j);	sätter in elementen i `[i,j)` i pos. `p`
s.erase(p);	tar bort elementet i position `p` från `s`
s.erase(p,p2);	tar bort elementen i `[p,p2)`från `s`

Iteratorer
Exempel: `vector::assign`, `vector::insert` och `vector::erase`

```
int a[] {1,2,3,4};
vector<int> v;
v.assign(a, a+4);
print_seq(v);          length = 4: [1][2][3][4]

v.insert(v.begin()+2, 3, 9);
print_seq(v);          length = 7: [1][2][9][9][9][3][4]

v.erase(v.begin()+5);
print_seq(v);          length = 6: [1][2][9][9][9][4]

v.erase(v.begin(),v.begin()+2);
print_seq(v);          length = 4: [9][9][9][4]
```


Avbildningar och mängder

Associativa containers

- Tabeller med söknycklar – t.ex. telefonlista med 2 kolumner (namn, telnr) där namnet utgör söknyckel
- Implementering i form av standardklasser

`map<Nyckel,Värde>`	Varje nyckel förekommer precis en gång
`multimap<Nyckel,Värde>`	En nyckel kan förekomma mer än en gång
`set<Nyckel>`	Varje nyckel förekommer precis en gång
`multiset<Nyckel>`	En nyckel kan förekomma mer än en gång

`set` *är i princip en* `map` *utan värden.*

Avbildningar och mängder
<set>: `std::set`

```
void test_set()
{
    std::set<int> ints{1,3,7};

    ints.insert(5);
    for(auto x : ints) {
        cout << x << " ";
    }
    cout << endl;
    auto has_one = ints.find(1);

    if(has_one != ints.end()){
        cout << "one is in the set\n";
    } else {
        cout << "one is not in the set\n";
    }
}
```

Eller

```
    if(ints.count(1))
```

```
1 3 5 7
one is in the set
```


Avbildningar och mängder
<map>: `std::map`

```
map<string, int> msi;
msi.insert(make_pair("Kalle", 1));
msi.emplace("Lisa", 2);
msi["Kim"]= 5;
for(auto& a: msi) {
    cout << a.first << " : " << a.second << endl;
}
cout << "Lisa --> " << msi.at("Lisa") << endl;
cout << "Hasse --> " << msi["Hasse"] << endl;
auto nisse = msi.find("Nisse");
if(nisse != msi.end()) {
    cout << "Nisse : " << nisse->second << endl;
} else {
    cout << "Nisse not found\n";
}
```

```
Kalle : 1
Kim : 5
Lisa : 2
Lisa --> 2
Hasse --> 0
Nisse not found
```

Avbildningar och mängder

En `std::set` *är i princip en* `std::map` *utan värden*

Operationer på `std::map`

```
insert, emplace, [], at, find, count,
erase, clear, size, empty,
lower_bound, upper_bound, equal_range
```

Operationer på `std::set`

```
insert, find, count,
erase, clear, size, empty,
lower_bound, upper_bound, equal_range
```


Läsanvisningar

Referenser till relaterade avsnitt i Lippman

Pekare och Referenser 2.3

Arrayer och pekare 3.5

Flerdimensionella arrayer 3.6

- Sekvenser 3.3, 9.1, 9.2, 9.3.1 – 9.3.3
- Iteratorer 3.4, 9.2.1

Associativa containers 11.1–11.3.2