EDAF30 – Programmering i C++

Typer, pekare

Sven Gestegård Robertz
Datavetenskap, LTH

2015

Innehåll

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
  - måste inte initialiseras när den definieras (odefinierat värde, som variabler)
  - har en adress
- fyra möjliga tillstånd
  1. pekar på ett objekt
  2. pekar på adressen omedelbart efter ett objekt
  3. pekar på ingenting: `nullptr`. Före C++11: `0`
  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
```

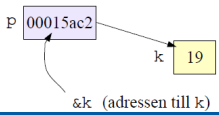
Referensoperatorn `&`, *address of*

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

Pekare till `int`

```
int *p1;
int *p2;
int n=4;
int m=5;

p1 = &n;   // p1 pekar på n
p2 = &m;   // p2 pekar på m
*p1 = 7;   // n blir 7
*p2 = *p1; // m blir också 7
p2 = p1;   // p2 pekar på n
```


specialfall: `void`-pekare

En `void*` kan peka på vad som helst (objekt av godtycklig typ.)

I C omvandlas `void*` implicit till/från varje pekartyp.

I C++ omvandlas `T*` implicit till `void*`. Åt andra hållet krävs en explicit *type cast*.

```
char i = 0;
char j = 0;

char* p = &i;
void* q = p;
int* pp = q; /* unsafe, legal C, not C++ */

*pp = -1;    /* overwrite memory starting at &i */
```

`const` och pekare

`const` modifierar allt som står till vänster om (undantag: om **`const`** står först modifieras det som kommer omedelbart efter)

```
void Exempel( int* ptr,
              int const * ptrToConst,
              int* const constPtr,
              int const * const constPtrToConst )
{
    *ptr = 0;                   // OK: ändrar innehållet
    ptr  = nullptr;             // OK: ändrar pekaren

    *ptrToConst = 0;            // Fel! kan inte ändra innehållet
    ptrToConst  = nullptr;      // OK: ändrar pekaren

    *constPtr = 0;              // OK: ändrar innehållet
    constPtr  = nullptr;        // Fel! kan inte ändra pekaren

    *constPtrToConst = 0;       // Fel! kan inte ändra innehållet
    constPtrToConst  = nullptr; // Fel! kan inte ändra pekaren
}
```


`const` och pekare
Kod-konventioner

C++: * skrivs ihop med typen

```
int*       ptr;
const int* ptrToConst;  //NB! (const int) *
int const* ptrToConst,  // ekvivalent, tydligare?

int *const  constPtr;   // pekaren är konstant

const int* const constPtrToConst; // Både pekare och värde
int const* const constPtrToConst; // ekvivalent, tydligare?
```

C: * skrivs ihop med namnet

Skilnader:

```
int        *ptr;
const int  *ptrToConst;
```


Pekare
Var tydlig med deklarationer

Se upp när du läser:

```
char *strcpy(char *dest, const char *src);

const char *ptrToConst; // (const char)*, inte const (char*)
```

Använd tumregeln: "en deklaration per rad"

```
int* a;     // pekare till int
int* a, b;  // a är pekare till int, b är int-variabel
int *a, *b; // a och b är pekare till int
```


Pekare

Pekare på konstanter och konstant pekare

```
int k;            // ändringsbar int
int const c = 100;// konstant int
int const * pc;   // pekare till konstant int
int *pi;          // pekare till ändringsbar int

pc = &c;   // OK
pc = &k;   // OK, men k kan inte ändras via *pc
pi = &c;   // FEL! pi får ej peka på en konstant
*pc = 0;   // FEL! pc pekar på en konstant

int* const cp = &k; // Konstant pekare
cp = nullptr;       // FEL! Pekaren ej flyttbar
*cp = 123;          // OK! Ändrar k till 123
```


Pekare

Sammanfattning

```
typ *p;     // p får typen
            // "pekare till typ"
p = &v;     // p tilldelas adressen till v

*p = 12;    // Det som p pekar på
            // tilldelas värdet 12
p1 = p2;    // p1 pekar på samma som p2
*p1 = *p2; // Det som p1 pekar på tilldelas
           // värdet som p2 pekar på
```


Pekare och referenser

Referenser till `int`

```
int dummy{100};
int n{4};
int m;          // Oinitierad variabel

int& r1 = n;    // r1 är en referens till n
r1 = 7;         // n får värdet 7
int& r2 = m;    // Referenser måste initieras
r2 = r1;        // m får värdet av n (==7)

int *p;         // Oinitierad pekare

p = &r1;        // p pekar på n (referenser har ingen adress)
*p = 4;         // n får värdet 4
r2 = *p + 8;    // m får värdet 12
r2 = *(p + 8); // !!! m får värdet 100 (just denna gång)
```

Pekare och referenser

Pekarversion och referensversion av `swap`

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

```
// Referensversionen
void swap(int& a, int& b)
{
   int tmp = a;
   a = b; b = tmp;
}
```

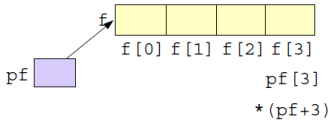
```
int m=3, n=4;
swap(m,n);    // Referensversionen används
swap(&m,&n); // Pekarversionen används
```


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]`)

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
```

Definition

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

`p + i*sizeof(T)`

Pekare och arrayer

Nollställning av en array `f[4]`

```
int f[4];
...
for (int i=0; i != 4; ++i)
   f[i] = 0;
```

```
float f[4];
...
for (float* p=f; p != f+4; ++p) // p är float-pekare
   *p = 0;
```

Nollställning av en array `f` – C++11

```
// Alt. 3 (C++11)
for (float& e : f)  // e är float-referens
   e = 0;
```


Pekare och arrayer

Funktion för nollställning av heltalsarrayer

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

Fungerar eftersom endast *adressen* värdekopieras medan *innehållet* ändras

Indexering av heltalsarrayer är oftast tydligare

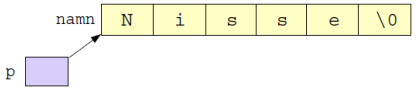
```
void zero(int x[], size_t n) {
   for (size_t i=0; i != n; ++i)
      x[i] = 0;
}
```


Pekare och textsträngar

Textsträngar i form av "array av `char`"

```
char namn[] = "Nisse";
char* p;
p = namn;
cout << p << endl;
cout << p+3 << endl;       // Skriver en delsträng  (typ: char*)
cout << *(p+3) << endl;    // Skriver en char       (typ: char)
cout << namn + 3 << endl; // Samma som p+3
cout << namn[3] << endl;  // ett tecken
```

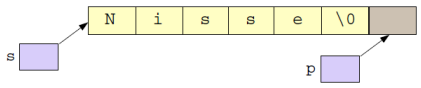
```
Nisse
se
s
se
s
```



Pekare och textsträngar

Puzzle Corner version av `strlen` (ger stränglängden)

```
// Användning av const char* för
// att garanterat inte ändra något
int strleng(const char *s) {
   const char *p; // Pekaren kan ändras!
   for (p = s; *p++; );
   return p-s-1;
}
```



Pekare och textsträngar

Exempel på array av pekare (ett datums veckodag)

```
char const * wdays[] = {"måndag", "tisdag", "onsdag",
        "torsdag", "fredag", "lördag", "söndag"};
int main(int argc, char* argv[]) {
   int y, m, d, ind;
   if (argc!=4) {
      cout <<"Anropa enligt mönstret " << argv[0]
           << "yyyy mm dd" << endl;
      return -1;
   }
   y=atoi(argv[1]); m=atoi(argv[2]); d=atoi(argv[3]);
   if (m<3) {m+=12; y--;} // Zeller's kongruens (1886)!
   ind = (d + 2*m + (3*(m+1))/5
         + y + y/4 - y/100 + y/400) % 7;
   cout << argv[1] << "-" << argv[2] << "-"
        << argv[3] << " är en " << wdays[ind] << endl;
} // Datumformat enligt standarden ISO8601
```


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

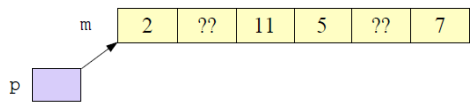
| **`int`** | **`int`** | **`int`** | **`int`** | **`int`** | **`int`** | **`int`** | **`int`** | **`int`** | **`int`** | **`int`** | **`int`** |

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

```
void addone(int (a*)[3]);   // Ett sätt att deklarera parametern

void addone(int a[][3]);    // Ett annat. tydligare?
{
   for (int i=0;i<2;i++) {
      for (int j=0;j<3;j++) {
         a[i][j]++;
      }
   }
}
void printmatr(int a[][3])
{
   for (int i=0;i<2;i++) {
      for (int j=0;j<3;j++) {
         cout << setw(8) << a[i][j] << " ";
      }
      cout << endl;
   }
}
```

Tabeller, matriser

Initiering och anrop

```
void addone(int a[][3]);
void printmatr(int a[][3]);

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

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

```
      1       2       3
      4       5       6
      7       8       9
-----------------
      1       2       3
      4       5       6
      7       8       9
```


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`

Exempel: pekare till `std::pair`

```
#include <utility>
...
pair<string, int> p1("Nils", 42);
cout << p1.first << " " << p1.second << endl;

pair<string, int>* pp = &p1;
pp->second = 43;
pp->first = "Nisse";
cout << p1.first << " " << p1.second << endl;
```


Funktionspekare

Pekare kan också peka på funktioner

```
float (*pf)(int,int);
float hypotenusa(int a, int b) {
   return sqrt((float)(a*a + b*b));
}
float medelv(int x, int y) {
   return ((float)(x + y))/2;
}
int main () {
   pf = hypotenusa;
   cout << pf(3,4) << endl;

   pf = medelv;
   cout << pf(3,4) << endl;
}
```


Funktionspekare

Funktionspekare kan vara argument till funktioner

```
float eval(float (*f)(int,int), int m, int n)
{
   return f(m, n);
}
float hypotenusa(int a, int b)
{
   return sqrt((float)(a*a + b*b));
}
float medelv(int x, int y)
{
   return ((float)(x + y))/2;
}
int main ()
{
   cout << eval(hypotenusa, 3, 4) << endl;
   cout << eval(medelv, 3, 4) << endl;
}
```


Funktionspekare
Alternativ i C++

Funktionspekare finns i C. I C++ finns även

- *funktor*: klass med **`operator()`**
- *lambda*: "anonym funktor"

Uppräkningstyper
C-stil

`enum`: en mängd namgivna värden

```
enum ans {JA, NEJ, KANSKE, VET_EJ};
enum colour {BLUE=2, RED=3, GREEN=5, WHITE=7};

// Deklaration av variabler
colour fgcol=BLUE;
colour bgcol=WHITE;
ans svar;

// Tilldelningar
fgcol=RED;
bgcol=GREEN;
svar = NEJ;
fgcol = KANSKE; // error: cannot convert 'ans' to 'colour'
svar = 2;       // error: invalid conversion from 'int' to 'ans'

bool dumt = (fgcol == svar); // Tillåtet, ger kanske en varning
```

Uppräkningstyper
C++: `enum class`

Problem med `enum`

Namnen "läcker" ut i omgivande *scope*.

```
enum eyes {brown,  green, blue};
enum traffic_light {red, yellow, green};
```

```
error: redeclaration of 'green'
```

C++:`enum class`

```
enum class EyeColour {brown,  green, blue};
enum class TrafficLight {red, yellow, green};

EyeColour e;
TrafficLight t;

e = EyeColour::green;
t = TrafficLight::green;
```


A propos "namn-läckage"

I stället för

```
using namespace std;
```

är det ofta bättre vara specifik:

```
using std::cout;
using std::endl;
```

jfr Java:

```
import java.util.*;

import java.util.ArrayList;
```


Uppräkningstyper
Kommentarer

- **`enum class`**
  - En **`enum class`** implementerar alltid
    - initiering, tilldelning och jämförelseoperatorer (t ex == och <)
    - andra operatorer kan implementeras
  - Har ingen implicit konvertering till **`int`**
- **`enum`**
  - Värdena *är* heltal
- Ha ett värde som betyder "fel" eller "oinitierad".
  - det första värdet, om möjligt
  - initiera alltid variabler, annars blir det *odefinierat*
- Använd **`enum class`** om möjligt

Uppräkningstyper
Initiering

Deklarationer

```
enum alternatives {ERROR, ALT1, ALT2};
enum class alternatives2 {ERROR, ALT1, ALT2};
```

Variablerna får väldefinierade värden

```
alternatives  a{};
alternatives  b{ALT1};

alternatives2 p{};
alternatives2 q{alternatives2::ALT1};
```

Variablerna kan få vilka värden som helst

```
alternatives  x;
alternatives2 y;
```


Sammanfattning

Vi har talat om

- Pekare
- En- och flerdimensionella arrayer
- Uppräkningstyper

Nästa föreläsning:

Vi kommer att studera

- *Type casts*
- Minnesallokering
- Objektorientering, Klasser