EDAF30 – Programmering i C++

8. Typomvandlingar. Klasser: operatorer och polymorfism.

Sven Gestegård Robertz
Datavetenskap, LTH

2017

Innehåll

Typomvandlingar (casting)
Implicita Typomvandlingar

Automatiska typomvandlingar

- Uttryck av typen $x \odot y$, för någon binär operator $\odot$
  Ex: `double + int ==> double`
  `float + long + char ==> float`
- Tilldelningar och initieringar: Värdet i högerledet konverteras till samma datatyp som i vänsterledet
- Konvertering av typen hos aktuella parametrar till typen för de formella parametrarna
- Villkor i `if`-satser, etc. ⇒ **`bool`**
- C-array ⇒ pekare (*array decay*)
- `0` ⇒ `nullptr` (tom pekare, i C++11, tidigare definierade man ofta konstanten `NULL`)

Typomvandlingar (*type casts*)
Explicita, namngivna typomvandlingar (C++-11)

- **`static_cast`**`<new_type> (expr)`
  - omvandlar mellan kompatibla typer (*kollar inte talområden*)
- **`reinterpret_cast`**`<new_type> (expr)`
  - inget skyddsnät, samma som C-stil
- **`const_cast`**`<new_type> (expr)` - lägger till eller tar bort **`const`**
- **`dynamic_cast`**`<new_type> (expr)` - används för pekare till klasser. Gör typkontroll vid *run-time*, som i Java.

Exempel

```
char c;              // 1 byte
int *p = (int*) &c;  // pekar på int: 4 bytes

*p = 5; //fel vid exekvering, stack-korruption

int *q = static_cast<int*> (&c); // kompileringsfel
```


Typomvandlingar (*casting*)
Explicita typomvandlingar, C-stil

Syntax i C och i C++, som i Java

`(typnamn)uttryck` , t ex `(float) 10`

- Stor risk att göra fel - använd namgivna typomvandligar
  - blir tydligare i koden, t ex **`const_cast`** kan bara ändra **`const`**
  - lätt att söka efter: casts är bland det första man tittar på när man letar fel
- Varning i GCC: `-Wold-style-casts`
- Vanlig i äldre kod

Alternativ syntax i C++

```
typnamn(uttryck)
```

`typnamn` måste vara *ett ord*,
d v s **`int`** `*(...)` eller **`unsigned long`**`(...)` är inte OK.

Typomvandlingar (*casting*)
Varnande exempel

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

struct Point3d :public Point{
     int z;
};
```

Point:

x:
y:

Point3d:

x:
y:
z:

Datatyper och variabler

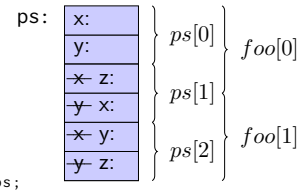
- Några begrepp:
  - en *typ* definierar mängden möjliga värden och operationer (för ett *objekt*)
  - ett *objekt* är ett stycke minne som innehåller ett *värde*
  - ett *värde* är en följd bitar som ska tolkas enligt en viss *typ*.

Typomvandlingar (*casting*)
Varnande exempel

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

Point ps[3];

struct Point3d{
     int x;
     int y;
     int z;
};
Point3d* foo = (Point3d*) ps;
```



Med *named casts* måste man använda **reinterpret_cast**<Point3d*>

med **static_cast** fås felet

```
invalid static_cast from type 'Point[3] to type 'Point3d*'
```


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*.

Överlagring av operatorer

Kan göras för de flesta operatorer, utom

```
sizeof   .   .*   ::   ?:
```

T ex kan dessa operatorer överlagras

```
=
+ - * / %
^ & | ~
<< >>
&& || !
!= == < >
++ -- += *= .......
() []
-> ->*
&
new delete new[] delete[]
```


Överlagring av operatorer

Överlagring av operatorer görs med syntaxen

```
returtyp operator⊗ (parametrar...)
```

för någon operator ⊗ t.ex. == eller +

Kan, för klasser, göras på två sätt:

- som medlemsfunktion
  - om ordningen på operanderna är lämplig
- som *fri* funktion
  - om det publika gränssnittet räcker, *eller*
  - om funktionen deklareras **friend**

Överlagring av operatorer
som medlemsfunktioner och fria funktioner

Exempel: deklaration som medlemsfunktioner

```
class Komplex {
public:
    Komplex(float r, float i) : re(r), im(i) {}
    Komplex  operator+(const Komplex& rhs) const;
    Komplex  operator*(const Komplex& rhs) const;
    // ...
private:
    float re, im;
};
```

Exempel: deklaration operator+ som friend

Deklaration inuti klassdefinitionen för Komplex:

```
friend Komplex operator+(const Komplex& l, const Komplex& r);
```

Notera antalet parametrar

Överlagring av operatorer

Överlagrade operatorer i användning:

Exempel: Komplexa tal

```
Komplex a = Komplex(1.2, 3.4);
Komplex b = Komplex(2.3, 1);
Komplex c = b;

a = b + c;        // a = b.operator+(c);
b = b + c * a;
c = a * b + Komplex(7, 4.5);
```


Överlagring av operatorer

Definition av operatorn + på två sätt

- Som medlemsfunktion

```
Komplex Komplex::operator+(const Komplex& rhs)const{
    Komplex temp;
    temp.re = re + rhs.re;
    temp.im = im + rhs.im;
    return temp;
}
```

- Som fri funktion

```
Komplex operator+(const Komplex& l, const Komplex& r){
        Komplex temp;
        temp.re = l.re + r.re;
        temp.im = l.im + r.im;
        return temp;
}
```

Att denna är friend syns bara i friend-deklarationen i klassen

Överlagring av operatorer

Definition av operatorn + på två sätt

- Som medlemsfunktion

```
Komplex Komplex::operator+(const Komplex& rhs)const{
    Komplex temp;
    temp.re = re + rhs.re;
    temp.im = im + rhs.im;
    return temp;
}
```

så att högra operanden inte kan ändras

så att vänstra operanden inte kan ändras

- Som fri funktion

```
Komplex operator+(const Komplex& l, const Komplex& r){
        Komplex temp;
        temp.re = l.re + r.re;
        temp.im = l.im + r.im;
        return temp;
}
```

Att denna är friend syns bara i friend-deklarationen i klassen

Överlagring av operatorer
Annan variant av + som använder +=

Klassdefinition

```
class Komplex {
public:
    const Komplex& operator+=(const Komplex& z) {
        re += z.re;
        im += z.im;
        return *this;
    }
    // ...
};
```

Returnerar **const**-referens för att inte tillåta t.ex. (a += b) = c;

Fri funktion, behöver inte vara `friend`

```
Komplex operator+(Komplex a, Komplex b) {
    return a+=b;
}
```

NB! *värdeanrop*: vi vill returnera *en kopia*.

Överlagring av operatorer

Binära operatorer: Variant av + med *heltal* som *höger* operand

Deklarationen (i klassdefinitionen av Komplex)

```
Komplex Komplex::operator+(int d) const;
```

Definitionen (utanför klassdefinitionen)

```
Komplex Komplex::operator+(int d) const {
        Komplex temp(*this);
        temp.re += d;
        return temp;
}
```


Överlagring av operatorer

Binära operatorer: Variant av + med heltal som *vänster* operand

- Problem: Kan inte använda medlemsfunktion! (varför?)

Deklarationen (Obs! *Utanför* klassdefinitionen)

```
Komplex operator+ (int d, const Komplex& v);
```

Definitionen (Obs! Ingen medlemsfunktion!)

```
Komplex operator+ (int d, const Komplex& v) {
        return v + d; // Utnyttjar andra +-op.!
}
```

Behöver inte vara **friend**: använder bara det publika gränssnittet.

Överlagring av operatorer
Exempel: <<

Exempel på friend-deklarerad operator: `<<` (`#include <ostream>`)

Deklarationen (i klassdefinitionen)

```
friend ostream& operator<<(ostream& o, const Komplex& v);
```

Definitionen (Obs! Ingen medlemsfunktion)

```
ostream& operator<<(ostream& o, const Komplex& v) {
    o << v.re << '+' << v.im << 'i';
    return o;
}
```


Överlagring av operatorer

Unära operatorer: Ökningsoperatorerna ++

Deklarationen (i klassdefinitionen)

```
const Komplex& operator++ ();    // preinkrement  (++v)
      Komplex  operator++ (int); // postinkrement (v++)
```

Dummy-parameter för att markera postinkrement-varianten

Definitionen (utanför klassdefinitionen)

```
const Komplex& Komplex::operator++ () { // prefix
        return (*this) += 1; // Returnera inkrementerad
}
Komplex Komplex::operator++ (int) {     // postfix
        Komplex temp(*this); // Kopia av detta objekt
        (*this) += 1;
        return temp; // Returnera oinkrementerad kopia
}
```


Typomvandlings-operatorer
Exempel: `Counter`

Konvertering till `int`

```
struct Counter {
    Counter(int c=0) :cnt{c} {};
    Counter& inc()       {++cnt; return *this;}
    Counter  inc() const {return Counter(cnt+1);}
    int get() const {return cnt;}
    operator int() const {return cnt;}
private:
    int cnt{0};
};
```

Notera: **`operator T()`**.

- returtyp anges inte
- kan deklareras **`explicit`**

Överlagring av operatorer
Tilldelningsoperatorn: **`operator=`** (*copy assignment*)

Deklarationen (i klassdefinitionen av Vektor)

```
const Vektor& operator=(const Vektor& v);
```

Definitionen (utanför klassdefinitionen)

```
const Vektor& Vektor::operator=(const Vektor& v)
{
  if (this != &v) {
      auto tmp = new int[sz];
      for (int i=0; i<sz; i++)
          tmp[i] = v.elem[i];
      sz = v.sz;
      delete[] elem;
      elem = tmp;
  }
  return *this;
}
```

1. kolla "self assignment"
2. Allokera nya resurser
3. Kopiera värden
4. Frigör gamla resurser

För felhantering bättre att allokera och kopiera först och bara göra **`delete`** *om allokeringen lyckades.*

Pekaren **`this`**
Självreferens

I en medlemsfunktion finns den implicita *pekaren* **`this`**, som pekar på objektet som funktionen anropades för. (jfr. **`this`** i Java).

Exempel på användning

```
struct Clock {
  Clock();
  Clock& set(int h, int m, int s);
  Clock& tick();
  Clock& print();
private:
  int seconds;
};
Clock& Clock::tick() {
    ++seconds;
    return *this;
}
```

Implicit deklarerad `Clock* const this`

Om vi har en variabel `Clock c;` kan vi nu "kedja" anrop:
`c.set(12,15,0).tick().tick().print();`

Polymorfism och dynamisk bindning

Polymorfism (mångformighet)

Överlagring	*Statisk bindning*
Generiska programenheter (templates)	*Statisk bindning*
Virtuella funktioner	*Dynamisk bindning*

Statisk bindning: Betydelsen hos en viss konstruktion avgörs vid *kompilering*

Dynamisk bindning: Betydelsen hos en viss konstruktion avgörs vid *exekvering*

Arv. Generalisering och specialisering

- Generalisering: abstrahera gränssnitt
- Specialisering: återanvändning och utökning

- Relationen *är*: En bil *är ett* fordon

Konkreta och abstrakta typer

Konkreta typer uppför sig "precis som inbyggda typer":

- *Representationen* ingår i *definitionen* [1]
- Kan placeras på stacken, och i andra objekt
- Kan refereras till direkt (och inte bara genom pekare eller referenser)
- Kan kopieras

Abstrakta typer isolerar användaren från implementationsdetaljer och *skiljer gränssnittet från representationen*:

- Representationen av objekt (*inkl. storleken!*) är okänd
- Kan bara refereras via pekare eller referenser
- Kan inte instansieras
  - endast konkreta subklasser

[1] kan vara privat, men är känd

Konkreta och abstrakta typer
En konkret typ: Vektor

```
class Vektor {
public:
    Vektor(int l = 10) :p{new int[l]},sz{l} {}
    ~Vektor() {delete[] elem;}
    int size() const {return sz;}
    int& operator[](int i) {assert(i<sz); return elem[i];}
private:
    int *elem;
    int sz;
};
```

Generalisering: *extract interface*

```
class Container {
public:
    int size() const;
    int& operator[](int o);
};
```


Konkreta och abstrakta typer
Generalisering: en abstrakt typ, Container

```
class Container {
public:
    virtual int size() const =0;
    virtual int& operator[](int o) =0;
    virtual ~Container() {}
};
```

- *pure virtual* funktion
- Abstrakt klass
- eller `interface` i Java

```
class Vektor :public Container {
public:
    Vektor(int l = 10) :p{new int[l]},sz{l} {}
    ~Vektor() {delete[] elem;}
    int size() const override {return sz;}
    int& operator[](int i) override {assert(i<sz); return elem[i];}
private:
    int *elem;
    int sz;
};
```

- `extends` (eller `implements`) `Container` i Java
- `override` motsvarar `@Override` i Java (C++11)
- En polymorf typ måste ha en virtuell destruktor

Konkreta och abstrakta typer
Användning av abstrakta klassen

```
  void fill(Container& c, int v)
{
    for(int i=0; i!=c.size(); ++i){
        c[i] = v;
    }
}
void print(Container& c)
{
    for(int i=0; i!=c.size(); ++i){
        cout << c[i] << " " ;
    }
    cout << endl;
}
void test_container()
{
    Vektor v(10);

    print(v);
    fill(v,3);
    print(v);
}
```


Konkreta och abstrakta typer
Användning av abstrakta klassen

Anta nu att vi har två andra subklasser till `Container`

```
class MyArray : public Container { ...};
class List : public Container { ...};

void test_container()
{
    Vektor v(10);
    print(v);
    fill(v);
    print(v);

    MyArray a(5);
    fill(a);
    print(a);

    List l{1,2,3,4,5,6,7};
    print(l);
}
```

- Dynamisk bindning av `Container::size()` och `Container::operator[]()`

Konkreta och abstrakta typer
Variant, utan att ändra Vektor

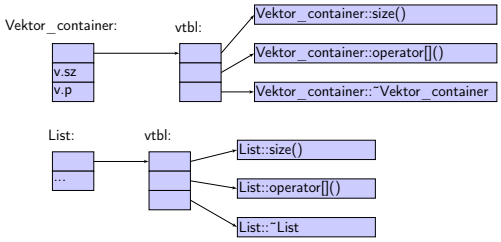
Om vi inte kan (eller vill) ändra klassen `Vektor` kan vi använda den för att skapa en ny klass:

```
class Vektor_container :public Container {
public:
    Vektor_container(int l = 10) :v{l} {}
    ~Vektor_container() =default;
    int size() const override {return v.size();}
    int& operator[](int i) override {return v[i];}
private:
  Vektor v;
};
```

- `Vektor` är en konkret klass
- Notera att `v` är ett `Vektor`-objekt, inte en referens
  - Skillnad från Java
- `Vektor`s destruktor (för `v`) anropas implicit

Dynamisk bindning

- virtuell funktions-tabell(*vtbl*)
  - innehåller pekare till objektets virtuella funktioner
  - varje klass med någon virtuell medlemsfunktion har en vtbl
  - varje objekt har en pekare till klassens vtbl
  - anrop av en virtuell funktion (typiskt) $< 25\%$ dyrare



Nästa föreläsning

Dynamisk poymorfism och arv kapitel 15 – 15.4
Tillgänglighet och scope kapitel 15.5 – 15.6
Typomvandling och arv kapitel 15.2.3
Arv och resurshantering 15.7
Polymorfa typer och containers 15.8
Multipelt arv 18.3
Virtuella basklasser 18.3.4 – 18.3.5

Läsanvisningar

Referenser till relaterade avsnitt i Lippman
Typomvandingar 4.11
Operatorer och typomvandling kapitel 14
Dynamisk poymorfism och arv kapitel 15 – 15.2.2
Pekaren `this` s. 257–260

Funktionspekare

Pekare kan också peka på funktioner

```
double hypotenuse(int a, int b) {
    return sqrt( a*a + b*b);
}

double add(int x, int y) {
    return x+y;
}

int main() {
    double (*pf)(int, int);

    pf = hypotenuse;
    cout << "hypotenuse: " << pf(3,4) << endl;

    pf = add;
    cout << "add: " << pf(3,4) << endl;
}
```


Funktionspekare

Funktionspekare kan vara argument till funktioner

```
double eval(double (*f)(int,int), int m, int n)
{
   return f(m, n);
}

double hypotenuse(int a, int b)
{
   return sqrt(a*a + b*b);
}
double add(int x, int y)
{
   return x + y;
}
int main ()
{
   cout << eval(hypotenuse, 3, 4) << endl;
   cout << eval(add, 3, 4) << endl;
}
```

Funktionspekare
Alternativ i C++

Funktionspekare finns i C. I C++ finns även

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

Pekare och textsträngar
C-strängar

Lagring i minnet

- *null-terminerad*: tecknet \0 markerar slutet på strängen
- `char namn[]= "Nils";  // längd = 5 bytes`

  är ekvivalent med

  `char namn[] = {'N', 'i', 'l', 's', '\0'};`
adress	data
namn	N
namn+1	i
namn+2	l
namn+3	s
namn+4	\0

Access av element

```
cout << namn[1] << namn[3];
```
```
is
```


Pekare och textsträngar
C-strängar

Notera returtyperna

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