EDAF30 – Programmering i C++

3. Mer om klasser. Funktionsanrop

Sven Gestegård Robertz
Datavetenskap, LTH

2016

Innehåll

1. Klasser
   - Konstruktorer
   - pekaren `this`
   - `const` för objekt och medlemmar
   - Kopiering
   - `friend`
   - `inline`
   - Statiska medlemmar
2. Funktionsanrop
   - Referens- eller värdeanrop, tumregler

Användardefinierade typer
Kategorier

- Konkreta klasser
- Abstrakta klasser
- Klasshierarkier

Användardefinierade typer
Konkreta klasser

En konkret typ

- kan användas på samma sätt som en inbyggd typ
- dess representation ingår i dess definition, så att objekt kan
  - placeras
    - på stacken (d v s i lokala variabler)
    - i andra objekt
    - i statiskt allokerat minne (i "globala variabler")
  - kopieras
    - tilldelning av variabel
    - värdeanrop av funktion
  - refereras direkt (inte bara genom referenser eller pekare)
  - initieras direkt och fullständigt (med en *konstruktor*)

Konstruktorer
Två sätt att skriva initiering av medlemmar

Med initierarlista i konstruktorn.

```
class Bar {
private:
    int value;
    bool flag;
public:
    Bar(int v, bool b) :value{v},flag{b} {}
};
```

Medlemmar kan ha en *default initializer*, i C++11:

```
class Foo {
public:
    Foo() {}
    Foo(int v) :value{v} {}
private:
    int value{0};
    bool flag = false;
};
```

Om en medlem både har *default initializer* och initierare i konstruktorn finns, används initieraren i konstruktorn.

Konstruktorer

Defaultkonstruktor

- Svarar mot en konstruktor utan parametrar
  - Får ha parametrar med defaultvärden
- Definieras automatiskt om *inte någon konstruktor alls* är definierad (i deklaration: `=default`, kan ej anropas om `=delete`)

Defaultkonstruktor med initierarlista.

```
class Bar {
public:
    Bar(int v=100, bool b=false) :value{v},flag{b} {}
private:
    int value;
    bool flag;
};
```

Anrop av defaultkonstruktor kan *inte göras med tomma parenteser*.

Konstruktorer
Defaultkonstruktor och parenteser

```
Bar b1;
Bar b2{};
Bar be();   // kompileringsfel!  "most vexing parse"
Bar b3(25); // OK

Bar* bp1 = new Bar;
Bar* bp2 = new Bar{};
Bar* bp3 = new Bar(); //OK
```


Default-konstruktor och initiering

- *automatiskt genererad* default-konstruktor (=**default**) initierar *inte alltid* medlemmar
  - *globala variabler* initieras till 0 (motsv)
  - *lokala variabler* initieras inte *Betyder nåt annat än i Java*

```
struct A { int x; };

int i; // i initieras till 0
A a;   // a.x initieras till  0

int main() {
    int j;          // j initieras inte
    int k = int(); // k initieras till 0
    int l{};        // l initieras till 0

    A b;            // b.x initieras inte
    A c = A();      // c.x initieras till 0
    A d{};          // d.x initieras till 0
}
```

- *använd alltid initierarlista*
- *implementera alltid default-konstruktor (eller =**delete**)*

Konstruktorer
Default-konstruktor

Default-värden på parametrar

- Om en konstruktor kan anropas utan parametrar, är det en default-konstruktor.

```
class KomplextTal {
public:
    KomplextTal(float x=1):re(x),im(0) {}
    //...
};
```

ger samma default-konstruktor som den explicita

```
    KomplextTal():re{1},im{0} {}
```


Konstruktorer
Delegering (C++11)

I C++11 kan en konstruktor anropa en annan (som `this(...)` i Java).

```
struct Test{
    int val;
    bool bar;

    Test(int v, bool b) :val{v},bar{b} {}

    Test(int v) :Test(v,true) {};       // delegering

    Test() :Test(0,false) {};           // delegering
};
```


Pekaren **this**
Självreferens

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

- *Mer om detta när vi studerat pekare*

Konstanta objekt

- **const** betyder "jag lovar att inte ändra denna"
- Objekt (variabler) kan vara deklarerade **const**
  - "jag lovar att inte ändra värdet på variabeln"
- Referenser kan vara deklarerade **const**
  - "jag lovar att inte ändra värdet på variabeln som refereras"
  - en **const**& kan referera till ett icke-**const** objekt
  - vanligt för funktionsparametrar
- Medlemsfunktioner kan vara deklarerade **const**
  - "jag lovar att funktionen inte ändrar objektets tillstånd"
  - *tekniskt: implicit deklaration* `const T* const this`

Konstanta objekt
Exempel

const objekt och **const** funktion

```
class Point{
public:
    Point(int xi, int yi) :x{xi},y{yi}{}
    int get_x() const {return x;}
    int get_y() const {return y;}
    void set_x(int xi) {x = xi;}
    void set_y(int yi) {y = yi;}
private:
    int x;
    int y;
};
void example(Point& p, const Point& o) {
    p.set_y(10);
    cout << "p: "<< p.get_x() << "," << p.get_y() << endl;

    o.set_y(10);
    cout << "o: "<< o.get_x() << "," << o.get_y() << endl;
}
passing 'const Point' as 'this' argument discards qualifiers
```


Konstanta objekt
Exempel

Notera **const** i deklarationen (och i definitionen!) av medlemsfunktionen `get_x()`: (“**const** *ingår i namnet*”)

```
class Point {
public:
    //...
    int get_x() const;      // funktionsdeklaration
    //...
};


int Point::get_x() const    // funktionsdefinition
{
    return x;
}
```


Konstanta objekt
Exempel: överlagring med **const**

Funktionerna `get_x()` och `get_x()` **const** *är olika funktioner*.

Exempel

```
struct Point {
    Point(int xi, int yi) :x{xi},y{yi}{}
    int& get_x()       {return x;}
    int  get_x() const {return x;}
    int& get_y()       {return y;}
    int  get_y() const {return y;}
private:
    int x;
    int y;
};
```

- Om `get_x()` anropas på ett
  - **const**-objekt fås en kopia
  - icke-**const**-object fås en referens

Användardefinierade typer
Konkreta klasser

En konkret typ

- kan användas på samma sätt som en inbyggd typ
- dess representation ingår i dess definition, så att objekt kan
  - placeras
    - på stacken (d v s i lokala variabler)
    - i andra objekt
    - i statiskt allokerat minne (i “globala variabler”)
  - kopieras
    - tilldelning av variabel
    - värdeanrop av funktion
  - refereras direkt (inte bara genom referenser eller pekare)
  - initieras direkt och fullständigt (med en *konstruktor*)

Konstruktorer

Kopieringskonstruktor

- Anropas vid initiering av ett objekt
- Anropas *inte* vid tilldelning
- Kan definieras, men i annat fall definieras automatiskt en standardvariant av kopieringskonstruktor (`=default`, `=delete`)

```
void function(Bar); // värdeanropad parameter

Bar b1(10,false};

Bar b2{b1};         // kopieringskonstruktorn anropas
Bar b3(b2);         // kopieringskonstruktorn anropas
Bar b4 = b2;        // kopieringskonstruktorn anropas

function(b2);       // kopieringskonstruktorn anropas

b4 = b3;            // kopieringskonstruktorn anropas inte
```


Kopieringskonstruktorer
default

- Deklaration:

```
class C {
public:
    C(const C&) =default;
};
```

- default-kopieringskonstruktor
  - Genereras automatiskt om den inte definieras i koden
    - undantag: om det finns medlemmar som inte kan kopieras
  - Grund kopiering av varje medlem
    - Fungerar för inbyggda typer
    - Fungerar för *klasser som uppför sig som inbyggda typer*
    - *Fungerar inte* för klasser som hanterar resurser “manuellt”

Konstruktorer
Specialfall: noll eller en parameter

Kopieringskonstruktor

- Har en **const** & som parameter: `Bar::Bar(const Bar& b);`

Typomvandlingskonstruktor

- En konstruktor med en parameter ger en implicit typkonvertering från parameterns typ

```
class KomplextTal {
public:
    KomplextTal():re{1},im{0} {}
    KomplextTal(const KomplextTal& k) :re{k.re},im{k.im} {}
    KomplextTal(double x):re{x},im{0} {}
    //...
private:
    double re;
    double im;
};
```

default constructor copy constructor converting constructor

Typomvandlingskonstruktor
Varning - implicit omvandling

```
class Vector{
public:
    Vector(int s); // create Vector with size s
    ...
    int size();    // return size of Vector
    ...
};

void example_vector()
{
    Vector v = 7;

    std::cout << "v.size(): " << v.size() << std::endl;

}
```

```
v.size(): 7
```

För `std::vector` är motsvarande konstruktor deklarerad

```
explicit vector( size_type count );
```


Typomvandlingskonstruktor och `explicit`

explicit gör att en konstruktor inte kan anropas implicit för typomvanding.

```
struct A
{
  A(int);
  // ...
};
```

```
struct B
{
  explicit B(int);
  // ...
};
```

```
A a1(2);       // OK
A a2 = 1;      // OK
A a3 = (A)1;   // OK

a3 = 17;       // OK [1]
```

```
B b1(2);       // OK
B b2 = 1;      // Fel! [2]
B b3 = (B)1;   // OK: explicit cast

b3 = 17;       // Fel! [3]
```

[1]: skapa en A(17), och därefter kopiera

```
[2]: conversion from 'int' to non-scalar type 'B' requested
[3]: no match for 'operator=' (operand types are 'B' and 'int')
```


Kopiering av objekt
Skillnad mellan *initiering* och *tilldelning*

```
void function(Bar); // värdeanropad parameter

Bar b1(10,false};

Bar b2{b1};          // kopieringskonstruktorn anropas
Bar b3(b2);          // kopieringskonstruktorn anropas
Bar b4 = b2;         // kopieringskonstruktorn anropas

function(b2);        // kopieringskonstruktorn anropas

b4 = b3;       // kopieringskonstruktorn anropas inte
```

Kopierings-tilldelning *(copy assignment)* – inte initiering

Kopiering av objekt
Tilldelningsoperatorn: **operator=** (*copy assignment*)

Tilldelningsoperatorn för kopiering (*copy assignment operator*) deklareras implicit

- med typen `T& T::operator=(const T&)`
- om ingen **operator=** för typen finns deklarerad
- om alla medlemsvariabler kan kopieras

Kopiering

- För "enkla" typer fungerar default-varianterna
- Mer om kopiering när vi studerar resurshantering

Konstruktorer
Initiering och tilldelning

Man *kan* (oftast) skriva som i Java, men det är inte lika effektivt.

Som i Java: tilldelning i konstruktor

```
class Foo {
public:
    Foo(const Bar& v) {
    value = v;  OBS! tilldelning, inte initiering
   }
private:
    Bar value; initieras innan konstruktorns kropp exekveras
};
```

Objektet är initierat innan konstruktorns funktionskropp anropas

Vänner (`friend`)

Funktioner eller klasser med full tillgång till medlemmar i en klass utan att själv vara medlem

Deklaration i klassen Point

```
class Point{
    //...
private:
    int x;
    int y;
    friend bool operator==(int, const Point&);
};
```

Definition utanför klassen Point

```
bool operator==(int x, const Point& p) {
    return p.x == x && p.y == 0;
}
```

Även medlemsfunktioner från andra klasser eller hela klasser kan vara vänner (jfr. package-synlighet i Java).

Klassdefinitioner
Medlemsfunktioner och `inline`

Inline-definition av en funktion

- Koden kan läggas ut direkt på det ställe anropet görs (inget funktionsanrop görs)
  - men kompilatorn måste inte göra det
- Lämpligt endast för mycket enkla funktioner
- Implicit om funktionsdefinitionen skrivs direkt i klassdefinitionen
- Om funktionen läggs utanför klassdefinitionen (via ::-notation) används nyckelordet `inline` framför

Klassdefinitioner
Medlemsfunktioner och `inline`, exempel

Inline i klassdefinitionen:

```
class Foo {
  public:
    int getValue() {return value;}
  // ...
};
```

Inline utanför klassdefinitionen:

```
inline int Foo::getValue()
{
    return value;
}
```


Statiska medlemmar

Statiska medlemmar: Delas mellan alla objekt i klassen

- *deklareras* i klassdefinitionen
- *definieras* utanför klassdefinitionen (om inte **const**)
- kan vara **public** eller **private**

Statiska medlemmar
Exempel: räkna allokeringar och avallokeringar

```
class Foo {
private:
    static int created;
    static int alive;
public:
    Foo() {++created; ++alive;}
    ~Foo() {--alive;}

    static void print_counts();
};
```

Definitioner: *NB! inte* **static**

```
int Foo::created{0};
int Foo::alive{0};

void Foo::print_counts()
{
    cout << alive << " / ";
    cout << created << endl;
}
```

```
void test_lifetimes()
{
    {
        Foo a;
        a.print_counts();

        Foo b;
        b.print_counts();
    }

    {
        Foo c;
        Foo::print_counts();
    }
    Foo::print_counts();
}
```

```
1 / 1
2 / 2
1 / 3
0 / 3
```

Statiska medlemmar
Exempel: räkna allokeringar och avallokeringar

```
class Foo {
private:
    static int created;
    static int alive;
public:
    Foo() {++created; ++alive;}
    ~Foo() {--alive;}

    static void print_counts();
};
```

Definitioner: *NB! inte* **static**

```
int Foo::created{0};
int Foo::alive{0};

void Foo::print_counts()
{
    cout << alive << " / ";
    cout << created << endl;
}
```

```
void test_lifetimes()
{
    {
        Foo a;
        a.print_counts();

        Foo b;
        b.print_counts();
    }

    {
        Foo c;
        Foo::print_counts();
    }
    Foo::print_counts();
}
```

```
1 / 1
2 / 2
1 / 3
0 / 3
```


Statiska medlemmar
Exempel: räkna allokeringar och avallokeringar

```
class Foo {
private:
    static int created;
    static int alive;
public:
    Foo() {++created; ++alive;}
    ~Foo() {--alive;}

    static void print_counts();
};
```

Definitioner: *NB! inte* **static**

```
int Foo::created{0};
int Foo::alive{0};

void Foo::print_counts()
{
    cout << alive << " / ";
    cout << created << endl;
}
```

```
void test_lifetimes()
{
    {
        Foo a;
        a.print_counts();

        Foo b;
        b.print_counts();
    }

    {
        Foo c;
        Foo::print_counts();
    }
    Foo::print_counts();
}
```

```
1 / 1
2 / 2
1 / 3
0 / 3
```


Statiska medlemmar
Exempel: räkna allokeringar och avallokeringar

```
class Foo {
private:
    static int created;
    static int alive;
public:
    Foo() {++created; ++alive;}
    ~Foo() {--alive;}

    static void print_counts();
};
```

Definitioner: *NB! inte* **static**

```
int Foo::created{0};
int Foo::alive{0};

void Foo::print_counts()
{
    cout << alive << " / ";
    cout << created << endl;
}
```

```
void test_lifetimes()
{
    {
        Foo a;
        a.print_counts();

        Foo b;
        b.print_counts();
    }

    {
        Foo c;
        Foo::print_counts();
    }
    Foo::print_counts();
}
```

```
1 / 1
2 / 2
1 / 3
0 / 3
```


Statiska medlemmar
Exempel: räkna allokeringar och avallokeringar

```
class Foo {
private:
    static int created;
    static int alive;
public:
    Foo() {++created; ++alive;}
    ~Foo() {--alive;}

    static void print_counts();
};
```

Definitioner: *NB! inte* **static**

```
int Foo::created{0};
int Foo::alive{0};

void Foo::print_counts()
{
    cout << alive << " / ";
    cout << created << endl;
}
```

```
void test_lifetimes()
{
    {
        Foo a;
        a.print_counts();

        Foo b;
        b.print_counts();
    }

    {
        Foo c;
        Foo::print_counts();
    }
    Foo::print_counts();
}
```

```
1 / 1
2 / 2
1 / 3
0 / 3
```


Fundera över – Objekt som returvärden

- Vi kan inte returnera referenser till lokala variabler (objekt som allokerats på stacken) i en funktion.
  - objektet förstörs vid **return** – *dangling reference*
- Hur (in)effektivt är det att istället returnera kopior av objekt?

Fundera över – Objekt som returvärden

- Vad skrivs ut av programmet nedan?

```
class C{
public:
    C() : a{0}, b{0} {cout << "A C was made.\n";}
    C(const C& o) :a{o.a},b{o.b} {cout << "A copy was made.\n";}
    void print() {cout << "C(" << a << "," << b << ")\n";}
private:
    int a;
    int b;
};

C f() {
    return C();
};

int main()
{
    cout << "Testing copying\n";
    C obj = f();
    obj.print();
}
```

Fundera över – Objekt som returvärden
Svar

Som programmet är skrivet anropas

- default-konstruktorn en gång och
- copy-konstruktorn två gånger
  1. vid initialiseringen av den anonyma returvariabeln i `f()` och
  2. vid initialiseringen av variabeln `obj`.
- Vad som faktiskt skrivs ut beror på hur bra kompilatorn är på att optimera!
- Både Visual Studio och gcc (CodeBlocks) optimerar bort bägge anropen av copy-konstruktorn och ger följande utskrift:

```
Testing copying
A C was made.
C(0,0)
```

- Andra kompilatorer kan ge andra svar

return value optimization (RVO)

Kompilatorn får lov att optimera bort kopiering av objekt vid **return** från funktioner

- *return by value* ofta effektivt, även för större objekt
- RVO tillåtet *även om copy-konstruktorn eller destruktorn har sidoeffekter*
- undvik sådana sidoeffekter för att göra koden portabel

Tumregler för funktionsparametrar

- Returnera värde oftare
- Överanvänd inte värdeanrop

"reasonable defaults"

	cheap to copy	moderately cheap to copy	expensive to copy
In	`f(X)`	`f(const X&)`	
In/Out	`f(X&)`		
Out	`X f()`		`f(X&)`

För resultat (returvärde), om kostnaden för kopiering är

- liten, eller måttlig ($< 1\ k$, sammanhängande): returnera värde (moderna kompilatorer gör RVO: return value optimization)
- stor : använd referensanrop som utparameter
  - eller kanske allokera med **new** och returnera pekare

Referens- eller värdeanrop
Tumregler

Om du skickar ett objekt som parameter till en funktion och

- du vill kunna *ändra* värdet på objektet
  - referens: **void** `f(T&);` eller
  - pekare: **void** `f(T*);`
- du vill *inte ändra* objektet, men det *är stort*
  - konstant referens: **void** `f(`**const** `T&);`
- annars, *använd värdeanrop*
  - **void** `f(T);`

Referens- eller värdeanrop
Tumregler

- Hur stort är "stort"?
  - mer än ett par *ord*
- När använda utparametrar?
  - skriv program som är lätta att förstå

Exempel: två funktioner:

```
void incr1(int& x)
{
   ++x;
}

int incr2(int x)
{
  return x + 1;
}
```

Användning:

```
int v = 0;
...

incr1(v);
...

v = incr2(v);
```

Här är det tydligare att
`v = incr2(v)` ändrar `v`

referens eller pekare?

- nödvändig/obligatorisk parameter: skicka referens
- valfri parameter: skicka pekare (kan vara `nullptr`)

```
void f(widget& w)
{
    use(w); //required parameter
}

void g(widget* w)
{
    if(w) use(w); //optional parameter
}
```

Läsanvisningar

Referenser till relaterade avsnitt i Lippman

Klasser	2.6, 7.1.4, 7.1.5
Konstruktorer	7.5–7.5.4
(Aggregate classes)	("C-structs"utan konstruktorer) 7.5.5
Destruktorer	13.1.3
const, constexpr	2.4
this och const	s 257–258
inline	6.5.2, s 273
friend	7.2.1
static members	7.6
Kopiering	13.1.1
Tilldelning	13.1.2
Operatoröverlagring	14.1