

TDDC76

Abstraktion, Klasser, överlagring och
testdriven utveckling

Eric Ekström & Klas Arvidsson

Institutionen för datavetenskap

- 1 Allmän info
- 2 Klasser
- 3 Filuppdelning
- 4 Inkludering och kompilering
- 5 Överlagring
- 6 Undantag
- 7 Testning

Allmän info

- Inget nytt

- 1 Allmän info
- 2 Klasser**
- 3 Filuppdelning
- 4 Inkludering och kompilering
- 5 Överlagring
- 6 Undantag
- 7 Testning

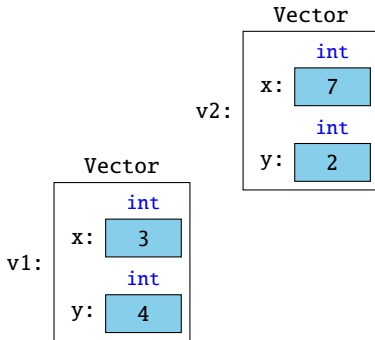
Klasser - repetition av aggregat

- En strukt är en egenskapad datatyp som består av flera andra data.

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

int main()
{
    Vector v1 {3, 4};
    Vector v2 {7, 2};

    cout << v1.x << endl;
    cout << v2.y << endl;
}
```



Klasser

Imperativt

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

int length(Vector const& v)
{
    return sqrt(v.x * v.x + v.y * v.y);
}

int main()
{
    Vector v {3, 4};
    cout << length(v) << endl;
}
```

Klasser

Imperativt

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

int length(Vector const& v)
{
    return sqrt(v.x * v.x + v.y * v.y);
}

int main()
{
    Vector v {3, 4};
    cout << length(v) << endl;
}
```

Objektorienterat

```
class Vector
{
public:
    int x;
    int y;

    int length() const
    {
        return sqrt(x * x + y * y);
    }
};

int main()
{
    Vector v {3, 4};
    cout << v.length() << endl;
}
```

Klasser

Ny terminologi:

- Klass
- Objekt / Instans
- Datamedlem (medlemsvariabel)
- Medlemsfunktion

Nya koncept:

- Synlighet och inkapsling: `public`, `private`
- Ändringsskydd på funktioner
- Hur vet vi vilken `x` och `y` det handlar om?

Objektorienterat

```
class Vector
{
public:
    int x;
    int y;

    int length() const
    {
        return sqrt(x * x + y * y);
    }
};

int main()
{
    Vector v {3, 4};
    cout << v.length() << endl;
}
```


Klasser

Klass

- En sammansatt datatyp
- Beskriver
 - hur en variabel skulle kunna se ut
 - vilken data variabeln ska ha
 - vilka funktioner som går att applicera på datan

```
class A
{
public:
    int x; char y;
    void foo();
};
```

Objekt

- En variabel
 - datatyp
 - namn
 - värde
- Har funktioner som opererar på datan

```
A a {1, 'c'};
a.foo();
```

Klasser - synlighet

- Innehållet i en klass kan placeras i:
 - `public` - synligt från annan programkod
 - `private` - endast synligt från funktioner som tillhör klassen
- Kompilatorn ser till att reglerna följs

```
class Vector
{
public:
    float length() const
    {
        return sqrt(x * x + y * y); // Ok
    }
private:
    int x;
    int y;
};

int main()
{
    Vector v {3, 5};
    v.x = 7; // Komplierar ej!
    cout << v.length() << v.x; // ???
}
```

Klasser - inkapsling

- Extern kod kan bara använda det som är publikt!
- Det är tydligt vad vi avser extern kod ska använda
- Vi kan ändra allt som är privat senare och övrig kod påverkas inte eftersom kompilatorn förbjudit att använda annat än publika medlemmar
- Vi kan förhindra att datamedlemmar får fel värden. Alla ändringar går kontrollerat via våra medlemsfunktioner

standard: Försök gör så mycket som möjligt privat!

Klasser - ändringsskydd

- Vi kan ange att en funktion inte kommer ändra på någon av objektets datamedlemmar
- Anges med `const` efter funktionsdeklarationen
- Avsaknat av `const` anger ATT funktionen ändrar objektet (även om den inte gör det)
- Förtydligar för den som skriver koden, den som använder koden, och kompilatorn kontrollerar.
- Viktigt för konstanta objekt (t.ex. `const&` på parametrar)

```
class Vector
{
public:
    float length() const;
};
```

Klasser - konstruktor

- En konstruktor är en speciell medlemsfunktion som anropas när ett objekt skapas
- Tanken är att vi i konstruktorn tilldelar datamedlemmarna sina värden
- Inget returvärde -> resultatet är ett skapat objekt
- Har alltid samma namn som klassen
- I medlemsinitieringslistan får datamedlemmarna sina värden

```
Vector(int init_x, int init_y)
: x { init_x }, y { init_y } // datamedlemsinitieringslista
{
  // Funktionsblock
}
```

Klasser - konstruktor

Varför inte bara sätta värden efter?

```
int main()
{
    Vector v {};
    v.x = 4;
    v.y = 5;
}
```

Klasser - konstruktor

Varför inte bara sätta värden efter?

```
int main()
{
    Vector v {};
    v.x = 4;
    v.y = 5;
}
```

Dåligt!

- Exponerar interna detaljer
- Ingen felkontroll
- Opraktiskt
- Bryter inkapsling

Klasser - konstruktor

- Kan vi ha flera olika konstruktörer?
- Vad händer om vi inte skapar en konstruktor?
 - Vi kommer alltid ha en konstruktor. Om vi inte anger en skapar kompilator en åt oss.

```
// Flera konstruktörer  
Vector();  
Vector(int x, int y);  
  
// default-konstruktor  
Vector() = default;
```


- 1 Allmän info
- 2 Klasser
- 3 Filuppdelning**
- 4 Inkludering och kompilering
- 5 Överlagring
- 6 Undantag
- 7 Testning

Filuppdelning

- Klassdeklaration i en headerfil (.h)

vector.h

```
class Vector
{
public:
    Vector(int x, int y);
    float length() const;
    void normalize();

private:
    int x;
    int y;
};
```

Filuppdelning

- Klassdeklaration i en headerfil (.h)
- Definition av medlemsfunktioner i en implementationsfil (.cc)

vector.h

```
class Vector
{
public:
    Vector(int x, int y);
    float length() const;
    void normalize();

private:
    int x;
    int y;
};
```

vector.cc

```
#include "vector.h"

float Vector::length() const
{
    return sqrt(x * x + y * y);
}

void Vector::normalize()
{
    float l { length() };
    x = x / l; y = y / l;
}
```

Filuppdelning

- Klassdeklaration i en headerfil (.h)
- Definition av medlemsfunktioner i en implementationsfil (.cc)
- main-funktion i ett huvudprogram (.cc)

main.cc

```
#include <iostream>
#include "vector.h"

int main()
{
    Vector v {3, 4};
    v.normalize();
    cout << v.length()
         << endl;
    return 0;
}
```

vector.h

```
class Vector
{
public:
    Vector(int x, int y);
    float length() const;
    void normalize();

private:
    int x;
    int y;
};
```

vector.cc

```
#include "vector.h"

float Vector::length() const
{
    return sqrt(x * x + y * y);
}

void Vector::normalize()
{
    float l { length() };
    x = x / l; y = y / l;
}
```

Filuppdelning

Varför lägger vi klassdeklarationen i en h-fil?

- h-filen blir ett gränssnitt som programmet (och programmeraren) kan vända sig till för att veta vad klassen kan göra.

Filuppdelning

Varför lägger vi klassdeklarationen i en h-fil?

- h-filen blir ett gränssnitt som programmet (och programmeraren) kan vända sig till för att veta vad klassen kan göra.
- Implementationsdetaljer är gömda. Vi ska inte behöva veta hur en klass fungerar för att använda den.

Filuppdelning

Varför lägger vi klassdeklarationen i en h-fil?

- h-filen blir ett gränssnitt som programmet (och programmeraren) kan vända sig till för att veta vad klassen kan göra.
- Implementationsdetaljer är gömda. Vi ska inte behöva veta hur en klass fungerar för att använda den.
- Vi undviker dubbla definitioner av medlemsfunktioner när vi inkluderar koden i andra filer.

- 1 Allmän info
- 2 Klasser
- 3 Filuppdelning
- 4 Inkludering och kompilering**
- 5 Överlagring
- 6 Undantag
- 7 Testning

Inkludering och kompilering

Hur får anropande programmet tillgång till definitionerna?

```
g++ main.cc  
/usr/bin/ld: /tmp/ccJZJjmm.o: in function 'main':  
/main.cc:12: undefined reference to 'Vector::length() const'  
collect2: error: ld returned 1 exit status
```

Inkludering och kompilering

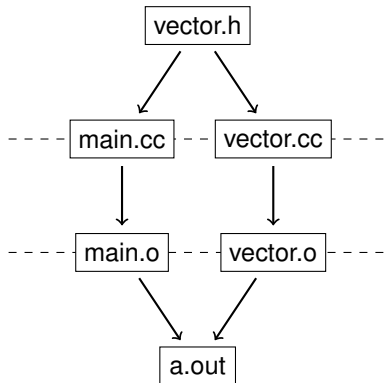
Hur får anropande programmet tillgång till definitionerna?

```
g++ main.cc  
/usr/bin/ld: /tmp/ccJZJjmm.o: in function 'main':  
/main.cc:12: undefined reference to 'Vector::length() const'  
collect2: error: ld returned 1 exit status
```

Vi måste kompilera vår implementationsfil också!

```
g++ main.cc vector.cc
```

Inkludering och kompilering



- Preprocessor - inkludering
- Kompilering - skapar kompillerade objekt-filer (.o)
- Länkning - länkar samman o-filer till ett körbart program

Inkludering och kompilering

- Alla cc-filer kompileras
- h-filer inkluderas
- Kompilera **aldrig** en h-fil!
 - Då genereras en .gch-fil som kompilator kommer använda i framtiden istället för er uppdaterade h-fil. Ta alltid bort filer med ändelse .gch!

Inkludering och kompilering

Vad händer om vi inkluderar samma fil två gånger?

```
using namespace std;
class Vector
{
    ...
};
```

```
#include "vector.h"
#include "vector.h"

int main()
{
    //..
}
```

Inkludering och kompilering

Vad händer om vi inkluderar samma fil två gånger?

```
using namespace std;
class Vector
{
    ...
};
```

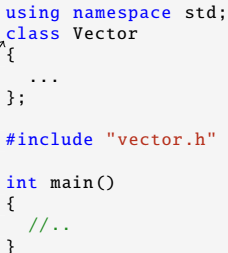
```
#include "vector.h"
#include "vector.h"

int main()
{
    //..
}
```

```
using namespace std;
class Vector
{
    ...
};

#include "vector.h"

int main()
{
    //..
}
```



Inkludering och kompilering

Vad händer om vi inkluderar samma fil två gånger?

```
using namespace std;
class Vector
{
    ...
};
```

```
#include "vector.h"
#include "vector.h"

int main()
{
    //..
}
```

```
using namespace std;
class Vector
{
    ...
};

#include "vector.h"

int main()
{
    //..
}
```

```
using namespace std;
class Vector
{
    ...
};

using namespace std;
class Vector
{
    ...
};

int main()
{
    //..
}
```

Inkludering och kompilering

Vad händer om vi inkluderar samma fil två gånger?

```
using namespace std;
class Vector
{
    ...
};
```

```
#include "vector.h"
#include "vector.h"

int main()
{
    //..
}
```

```
using namespace std;
class Vector
{
    ...
};

#include "vector.h"

int main()
{
    //..
}
```

```
using namespace std;
class Vector
{
    ...
};

using namespace std;
class Vector
{
    ...
};

int main()
{
    //..
}
```

KOMPILERAR EJ!

Inkludering och kompilering

Vi behöver en header guard!

```
#ifndef VECTOR_H
#define VECTOR_H

class Vector
{
    ...
};

#endif
```

- Alla h-filer **ska** ha en header guard.
- Använd aldrig `using namespace std;` i en h-fil. Då tvingas alla som inkluderar vår fil att också `using namespace std;`

- 1 Allmän info
- 2 Klasser
- 3 Filuppdelning
- 4 Inkludering och kompilering
- 5 Överlagring**
- 6 Undantag
- 7 Testning

Operatoröverlagring

Hur kan vi addera två vektorer och skriva ut resultatet?

```
Vector v1 {3, 4};  
Vector v2 {2, 5};  
  
Vector v3 { v1 + v2 };  
  
cout << v3 << endl;
```

KOMPILERAR EJ!

Det finns ingen operator+ eller operator<< för vår egen datatyp Vector.

Vi kan skapa dessa själva!

Operatoröverlagring

```
x = 10 + 20;  
// ekvivalent med  
x = operator+(10, 20);  
  
Vector v3 { v1 + v2 };  
Vector v3 { operator+(v1, v2) };
```

Hur skulle funktionsdeklarationen för operator+ se ut i vårt fall?

Operatoröverlagring

```
x = 10 + 20;  
// ekvivalent med  
x = operator+(10, 20);  
  
Vector v3 { v1 + v2 };  
Vector v3 { operator+(v1, v2) };
```

Hur skulle funktionsdeklarationen för operator+ se ut i vårt fall?

```
Vector operator+(Vector const& lhs, Vector const& rhs)  
{  
    // Implementera vektor-addition!  
}
```

Operatoröverlagring

Det går också att anropa en operator som en medlem:

```
// samma effekt  
v1 + v2;  
operator+(v1, v2);  
v1.operator+(v2);
```

Hur skulle vår funktionsdeklaration se ut om det var en medlemsfunktion?

Operatoröverlagring

Det går också att anropa en operator som en medlem:

```
// samma effekt  
v1 + v2;  
operator+(v1, v2);  
v1.operator+(v2);
```

Hur skulle vår funktionsdeklaration se ut om det var en medlemsfunktion?

```
Vector Vector::operator+(Vector const& rhs) const  
{  
    // Implementera vektor-addition!  
}
```

Var är lhs? Varför är funktionen const?

Operatoröverlagring

Två sätt att skriva en operator:

```
[returtyp] operator[symbol]([left], [right])  
{  
  // Instruktioner  
  [retursats]  
}
```

```
[returtyp] [klass]::operator[symbol]([right])  
{  
  // Instruktioner  
  [retursats]  
}
```


Operatoröverlagring

Vilken variant ska man välja när?

- Båda är okej. Välj den ni vill, men var så konsekventa det går!
- Det finns dock stunder då vi tvingas välja alternativ 1:
 - Om operanden på vänster sida är av en typ vi inte kan styra över.

```
Vector v {3, 4};  
v = 2 * v; // 2.operator*(v) fungerar ej!
```

Operatoröverlagring

På [cppreference](#) kan vi se hur operatorer ska vara implementerade.

Övning: Hur ser funktionsdeklarationerna ut för de två olika fallen av `operator<<`? Vilken av dem är genomförbar?

```
Vector v1 {3, 4};  
Vector v2 {2, 5};  
  
cout << v1 << v2 << endl;
```

Operatoröverlagring

På [cppreference](#) kan vi se hur operatorer ska vara implementerade.

Övning: Hur ser funktionsdeklarationerna ut för de två olika fallen av `operator<<`? Vilken av dem är genomförbar?

```
Vector v1 {3, 4};  
Vector v2 {2, 5};  
  
cout << v1 << v2 << endl;
```

```
ostream& ostream::operator<<(Vector const& rhs);  
ostream& operator<<(ostream& os, Vector const& rhs);
```

Funktionsöverlagring

- I C++ kan flera funktioner ha samma namn
- Typerna på och antal parametrar avgör vilken som anropas
- Kompilatorn väljer den som matchar

```
int triangle_area(int base, int height);           // v1
int triangle_area(int sidel, int side2, int side3); // v2
int triangle_area(int sidel, int side2, double angle); // v3
```

```
cout << triangle_area(1, 1);           // v1 anropas
cout << triangle_area(1, 1, 1);        // v2 anropas
cout << triangle_area(1, 1, 1.0);      // v3 anropas
```

Funktioner - default-argument

Ibland vill vi att en parameter alltid ska få ett visst värde, om vi inte anger något annat.

- vi sätter ett default-argument på den parametern
- parametrar med default-argument måste ligga sist i parameterlistan
- default-argument anges endast i deklARATIONEN av en funktion. Ej vid definition

```
void setfill(char fill_char = ' ');
```

```
setfill('*'); // fyll ut med '*'  
setfill();   // fyll ut med ' '
```

- 1 Allmän info
- 2 Klasser
- 3 Filuppdelning
- 4 Inkludering och kompilering
- 5 Överlagring
- 6 Undantag**
- 7 Testning

Undantag

Hur ska en programkomponent signalera till en annan att de värden komponenten fått att arbeta med inte är användbara?

- I första hand, se till att programmet inte kompilerar!
 - Välj bra datatyper så att det endast går att skicka in korrekt data.
- I andra hand, kasta ett undantag!

Undantag - kasta

Om ett undantag kastas avbryts den nuvarande funktionen och den anropande funktionen får en notis om att ett undantag kastades. Denna måste då fånga undantaget eller kasta om undantaget.

```
throw std::logic_error("Ett meddelande");  
  
throw std::runtime_error("Ett meddelande");
```

Om ett undantag kastas i main() avbryts körningen:

```
terminate called after throwing an instance of 'std::logic_error'  
what(): Invalid card value Aborted (core dumped)
```


Undantag - fånga

“Kör instruktionerna i block1. Om det där kastas ett undantag, avbryt block1 och kör instruktionerna i block2.”

```
int main()
{
    try
    {
        // block1
    }
    catch(std::exception& e) //Fångar alla standard-undantag
    {
        // block2
    }
}
```

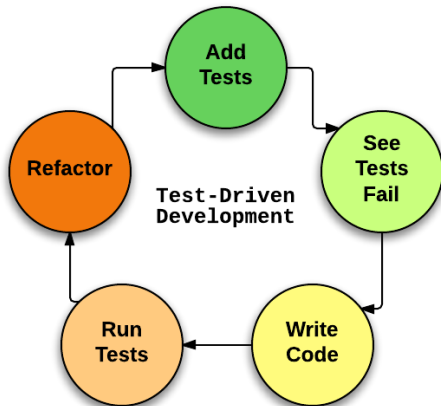
Undantag - exempel

```
int factorial(int i)
{
    if (i < 0)    throw std::logic_error("can't compute that!");
    if (i <= 1)  return 1;
    else        return factorial(i - 1) * i;
}
```

```
int main()
{
    try
    {
        int x { factorial(3) };
        int y { factorial(-5) };
        cout << y << endl;
    }
    catch(std::exception& e)
    {
        std::cout << "Error: " << e.what() << std::endl;
    }
}
```

- 1 Allmän info
- 2 Klasser
- 3 Filuppdelning
- 4 Inkludering och kompilering
- 5 Överlagring
- 6 Undantag
- 7 Testning**

Testdriven utveckling (TDD)



Catch

```
#include "catch.hpp"
#include <stdexcept>

int factorial(int i)
{
    if (i < 0)    throw std::logic_error("can't compute that!");
    if (i <= 1)  return 1;
    else        return factorial(i - 1) * i;
}

TEST_CASE( "Test factorial() function" )
{
    CHECK( factorial(1) == 1 );
    CHECK( factorial(3) == 6 );

    CHECK_FALSE( factorial(1) == 2 );

    CHECK_THROWS( factorial(-1) );
}
```

Testa med Catch

- Behöver testramverket catch:
 - `catch.hpp`, `test_main.cc`
 - Finns i givna filer på kurshemsidan.
- Behöver testfil och tillhörande kod som ska testas.

Kompilera koden som vanligt.

*.cc och *.cpp kompileras. *.h, *.hh och *.hpp inkluderas!

```
w++17 test_main.cc kod_test.cc kod.cc  
./a.out
```

```
=====
```

```
All tests passed(89 assertions in 14 test cases)
```

Snabbare kompilering

Kompilera `test_main.cc` separat. Detta görs bara en gång:

```
w++17 -c test_main.cc
```

- `-c` utför endast kompilering, ej länkning. Genererar en objektfil

Kompilera övriga filer och länka in `test_main.o`:

```
w++17 test_main.o kod_test.cc kod.cc
```

www.liu.se