TDIU20

Kursupplägg och Objektorientering

Eric Ekström

Institutionen för datavetenskap

LINKÖPING UNIVERSITY

1 Kursinformation
2 Klasser
3 Filuppdelning
4 Inkludering och kompilering
5 Undantag
6 Testning
7 Objektorientering - Exempel
8 Överlagring
9 Operatoröverlagring - Exempel
10 Övrigt (i mån av tid)

LINKÖPING UNIVERSITY

TDIU20 - Lärandemål

Studenten ska tillägna sig kunskaper om och färdigheter i objektorienterad programmering i programspråket C++.

Efter genomgången kurs ska studenten kunna:

- använda designprinciper, metoder och tekniker som används inom objektorienterad programmering för att konstruera program som löser givna problem.

TDIU20 - Lärandemål

Erics tolkning!

Skriva kod som

- är lätt att använda
- går att underhålla
- går att bygga vidare på
- främjar enklare samarbete

TDIU20 - Kursinnehåll

- 6 st föreläsningar
  - 1-2 Objektorientering
  - 3-4 Pekare och dynamiskt minne
  - 5 Arv och Polymorfi
  - 6 Tentaföreläsning
- 3 laborationer
- 3 lektioner
- Tentamen

All information finns på kurshemsidan:
https://www.ida.liu.se/~TDIU20/

TDIU20 - Personal

- Examinator - Klas Arvidsson
- Kursledare - Eric Ekström
- Kursassistent - Love Arreborn
- Assistenter
  - Nils Jakobsson
  - Wiktor Liew
  - Love Arreborn
  - Oliver Brandett

Kontaktuppgifter på kurshemsidan!

TDIU20 - Komma i mål

- Följ kursens alla moment
- Hjälp varandra (vad hjälper och vad stjälper?)
- Fråga kursledare och examinator (det är därför vi är här!)
- Investera den nödvändiga tiden
  - 4hp motsvarar ca 106h
  - Kursen använder periodens 7 första veckor (15h per vecka)
  - Vad (i alla kurser) spara du till vecka 8 och tentaperioden?

TDIU20 - Examination

- **LAB1** - Laborationer (3hp)
  3 uppgifter med redovisning och kodinlämning
- **DAT2** - Datortentamen (1hp)
  Ger slutbetyget i kursen

TDIU20 - Laborationer

- Webreg för registrering och resultat
  - Anmälan senast första labbpasset
- Arbeta i par (båda ansvariga för all er kod)
- Redovisning, kodinlämning och komplettering
- Deadlines i Timeedit
- Bonus för redovisning innan deadline och väl genomförd komplettering

TDIU20 - Datortentamen

- Liverättad tentamen i SU-salarna
- Studentklient för frågor och inlämning
- Möjlighet att komplettera under tentamens gång.

Tentamen består av två delar:

- **Del I:** fyra uppgifter. För godkänt måste alla vara avklarade.
- **Del II:** två uppgifter med 3 poäng vardera.
  Betyg *N* kräver *N* poäng på del II.

TDIU20 - Bonussystem

Bonusuppgifter

- Varje laboration har en bonusuppgift.
- Tillgodoräkna motsvarande uppgift i del I på tentamen.

Engagemangsbonus

- 3 lektioner, 3 labbdeadlines, 3 kompletteringar att genomföra snyggt
- 5 / 9 engagemang ger 1 bonuspoäng på del II på tentamen

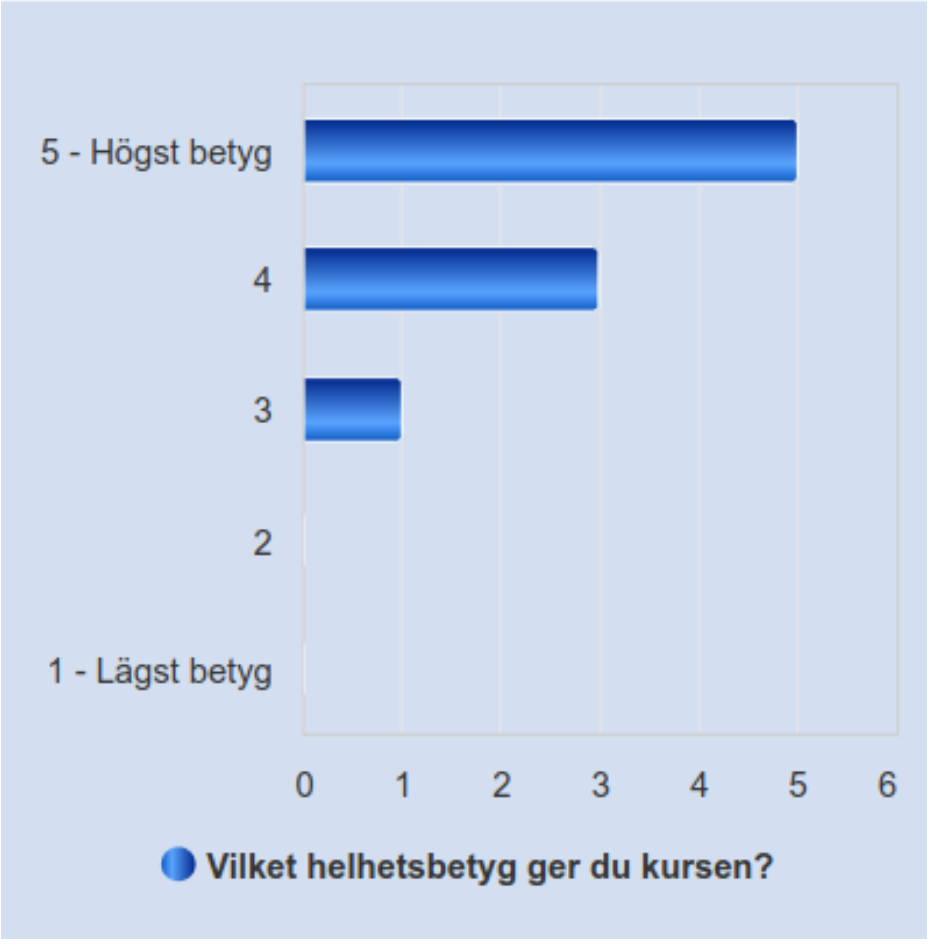
TDIU20 - Återkoppling från tidigare år

- 9.8 % svarsfrekvens
- 4 fritextsvar

- Önskas fler labbtillfällen.
- Labbassistenter var mycket uppskattade.

- Säg vad ni tycker! Svara på kursutvärderingen!

Inga ändringar görs till kursomgång 2025.



1 Kursinformation
2 Klasser
3 Filuppdelning
4 Inkludering och kompilering
5 Undantag
6 Testning
7 Objektorientering - Exempel
8 Överlagring
9 Operatoröverlagring - Exempel
10 Övrigt (i mån av tid)

Klasser

```
struct Rectangle
{
  int width;
  int height;
};

double area(Rectangle const& r)
{
  return r.width * r.height;
}


int main()
{
  Rectangle r {3, 4};
  cout << area(r) << endl;
}
```

Klasser

```
struct Rectangle
{
  int width;
  int height;
};

double area(Rectangle const& r)
{
  return r.width * r.height;
}


int main()
{
  Rectangle r {3, 4};
  cout << area(r) << endl;
}
```

```
class Rectangle
{
public:
  int width;
  int height;

  double area() const
  {
    return width * height;
  }
};

int main()
{
  Rectangle r {3, 4};
  cout << r.area() << endl;
}
```

Klasser - synlighet

```
class Rectangle
{
public:
  float area() const
  {
    // Ok
    return width * height;
  }
private:
  int width;
  int height;
};

int main()
{
  Rectangle r {3, 5};

  cout << r.area(); // A
  cout << r.width;  // B
  r.width = 7;      // C
}
```

Medlemmarna kan placeras i:

- `public`
- `private`
- (`protected`)

övning: Vilka av {A, B, C} kompilerar?

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 förbjuder användning av 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.

```
class Rectangle
{
public:
  float area() const;
};
```

Klasser - konstruktor

En konstruktor är en speciell medlemsfunktion som anropas när ett objekt skapas.

```
class Rectangle
{
public:
  Rectangle(int init_width, int init_height)
    // datamedlemsinitieringslista
    : width { init_width }, height { init_height }
  {
    // Funktionsblock
  }

private:
  int width;
  int height;
};
```

Klasser - konstruktor

Varför inte bara sätta värden direkt efter?

```
int main()
{
  Rectangle r {};
  r.width = 4;
  r.height = 5;
}
```

Klasser - konstruktor

Varför inte bara sätta värden direkt efter?

```
int main()
{
  Rectangle r {};
  r.width = 4;
  r.height = 5;
}
```

Dåligt!

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

Klasser - konstruktor

- Kan vi ha flera olika konstruktorer?
- Vad händer om vi inte skapar en konstruktor?

```
class Rectangle
{
public:
  // Flera konstruktorer
  Rectangle();
  Rectangle(int width, int height);

  // default-konstruktor
  Rectangle() = default;
};
```

1 Kursinformation
2 Klasser
3 Filuppdelning
4 Inkludering och kompilering
5 Undantag
6 Testning
7 Objektorientering - Exempel
8 Överlagring
9 Operatoröverlagring - Exempel
10 Övrigt (i mån av tid)

Filuppdelning

Filuppdelning

rectangle.h

```
class Rect
{
public:
  Rect(int w, int h);
  float area() const;
  bool square() const;

private:
  int width;
  int height;
};
```

Filuppdelning

rectangle.h

```
class Rect
{
public:
  Rect(int w, int h);
  float area() const;
  bool square() const;

private:
  int width;
  int height;
};
```

rectangle.cc

```
#include "rectangle.h"

float Rect::area() const
{
  return width * height;
}

void Rect::square() const
{
  return width == height;
}
```

Filuppdelning

main.cc

```
#include <iostream>

#include "rectangle.h"

int main()
{
  Rect r {3, 4};
  cout << r.square()
       << r.area()
       << endl;
  return 0;
}
```

rectangle.h

```
class Rect
{
public:
  Rect(int w, int h);
  float area() const;
  bool square() const;

private:
  int width;
  int height;
};
```

rectangle.cc

```
#include "rectangle.h"

float Rect::area() const
{
  return width * height;
}

void Rect::square() const
{
  return width == height;
}
```

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 Kursinformation
2 Klasser
3 Filuppdelning
4 Inkludering och kompilering
5 Undantag
6 Testning
7 Objektorientering - Exempel
8 Överlagring
9 Operatoröverlagring - Exempel
10 Övrigt (i mån av tid)

Inkludering och kompilering

Hur får anropande programmet tillgång till definitionerna?

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

Inkludering och kompilering

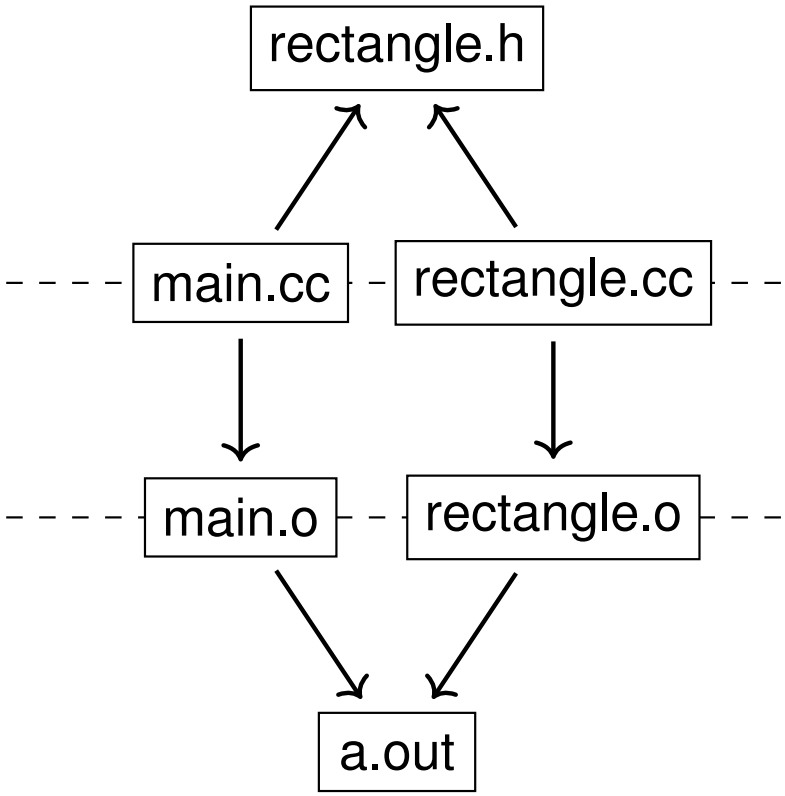
Hur får anropande programmet tillgång till definitionerna?

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

Vi måste kompilera vår implementationsfil också!

```
$ g++ main.cc rectangle.cc
```

Inkludering och kompilering



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

Inkludering och kompilering

- `.cc`-filer kompileras
- `.h`-filer inkluderas
- Kompilera **aldrig** en h-fil!

Inkludering och kompilering

Vad händer om vi inkluderar samma fil två gånger?

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

```
#include "rectangle.h"
#include "rectangle.h"

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

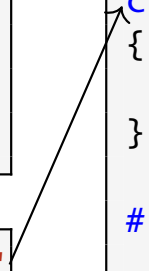
Inkludering och kompilering

Vad händer om vi inkluderar samma fil två gånger?

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

```
#include "rectangle.h"
#include "rectangle.h"

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



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

#include "rectangle.h"

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

Inkludering och kompilering

Vad händer om vi inkluderar samma fil två gånger?

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

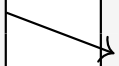
```
#include "rectangle.h"
#include "rectangle.h"

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

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

#include "rectangle.h"

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



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

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

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

Inkludering och kompilering

Vad händer om vi inkluderar samma fil två gånger?

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

```
#include "rectangle.h"
#include "rectangle.h"

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

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

#include "rectangle.h"

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

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

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

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

KOMPILERAR EJ!

Inkludering och kompilering

Vi behöver en header guard!

```
#ifndef RECTANGLE_H
#define RECTANGLE_H

class Rectangle
{
  ...
};

#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 Kursinformation
2 Klasser
3 Filuppdelning
4 Inkludering och kompilering
5 Undantag
6 Testning
7 Objektorientering - Exempel
8 Överlagring
9 Operatoröverlagring - Exempel
10 Övrigt (i mån av tid)

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(): Ett meddelande
Aborted (core dumped)
```

Undantag - fånga

"Kör instruktionerna i block1. Om det där kastas ett undantag, avbryt block1 och kör instruktionerna i block3. Då körs inte block2."

```
int main()
{
  try
  {
    // block1
    // block2
  }
  catch(std::exception& e) //Fångar alla standard-undantag
  {
    // block3
  }
}
```

Undantag - exempel

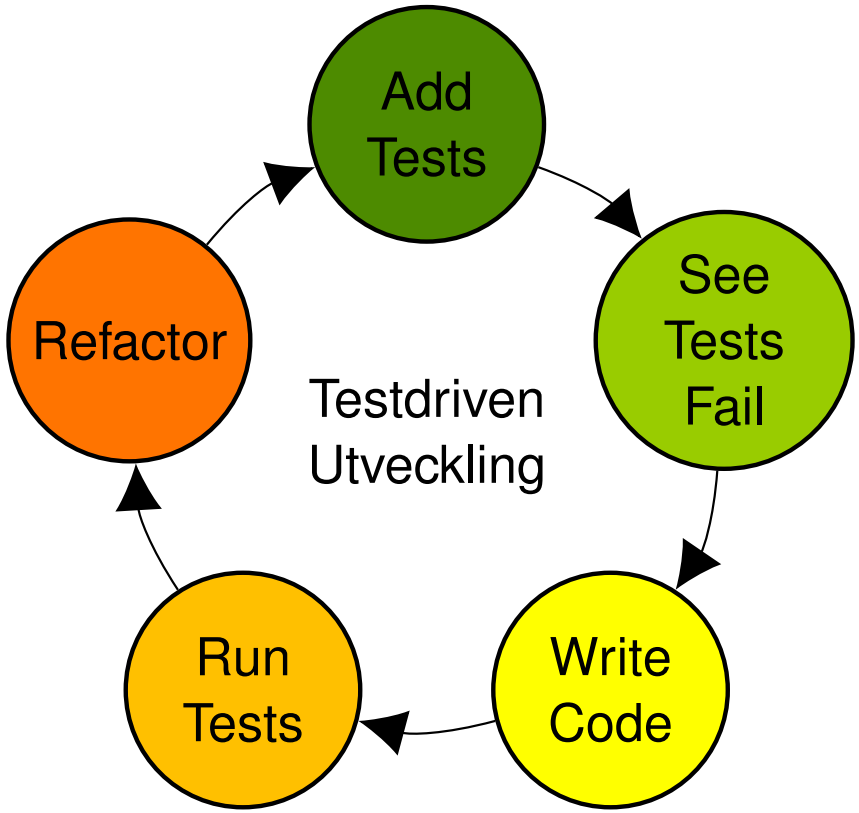
```
int factorial(int i)
{
  if (i < 0)  throw std::logic_error("I can't compute that!");
  if (i <= 1) return 1;
  else        return factorial(i - 1) * i;
}
```

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

1 Kursinformation
2 Klasser
3 Filuppdelning
4 Inkludering och kompilering
5 Undantag
6 Testning
7 Objektorientering - Exempel
8 Överlagring
9 Operatoröverlagring - Exempel
10 Övrigt (i mån av tid)

LINKÖPING UNIVERSITY

Testdriven utveckling (TDD)



Catch

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

int factorial(int i)
{
  if (i < 0)  throw std::logic_error("I 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
```

1 Kursinformation
2 Klasser
3 Filuppdelning
4 Inkludering och kompilering
5 Undantag
6 Testning
7 Objektorientering - Exempel
8 Överlagring
9 Operatoröverlagring - Exempel
10 Övrigt (i mån av tid)

LINKÖPING UNIVERSITY

Funktionsöverlagring

- I C++ kan flera funktioner ha samma namn
- Antal parametrar och deras typer avgör vilken som anropas
- Kompilatorn väljer den som matchar

```
int triangle_area(int base, int height);                // v1
int triangle_area(int side1, int side2, int side3);     // v2
int triangle_area(int side1, 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.

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

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

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;
```

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!

Operatoröverlagring

Hur fungerar strängaddition? Hur vet kompilatorn vad som ska göras?

```
string s1 {"c++ är "};
string s2 {"bäst!"};

string s3 { s1 + s2 };
cout << s3 << endl;
```

Operatoröverlagring

Hur fungerar strängaddition? Hur vet kompilatorn vad som ska göras?

```
string s1 {"c++ är "};
string s2 {"bäst!"};

string s3 { s1 + s2 };
cout << s3 << endl;
```

Det finns en funktion!

```
string operator+(string const& lhs, string const& rhs)
{
  // slå ihop två strängar
}
```

Operatoröverlagring

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

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

Operatoröverlagring

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

övning: 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 skapa en operator som en medlem:

```
v1 + v2;
// ekvivalent med
v1.operator+(v2);
```

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

Operatoröverlagring

Det går också att skapa en operator som en medlem:

```
v1 + v2;
// ekvivalent med
v1.operator+(v2);
```

övning: 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);
```

Operatoröverlagring

Hur ska vi använda operatoröverlagring?

- Använd operatoröverlagring om det
  - är naturligt och väntat
  - inte har några bieffekter
- Använd endast en operator till sitt tänkta ändamål
- Härma beteendet av andra operatorer i C++

1 Kursinformation
2 Klasser
3 Filuppdelning
4 Inkludering och kompilering
5 Undantag
6 Testning
7 Objektorientering - Exempel
8 Överlagring
9 Operatoröverlagring - Exempel
10 Övrigt (i mån av tid)

1 Kursinformation
2 Klasser
3 Filuppdelning
4 Inkludering och kompilering
5 Undantag
6 Testning
7 Objektorientering - Exempel
8 Överlagring
9 Operatoröverlagring - Exempel
10 Övrigt (i mån av tid)

Mer om datamedlemmar

- Datamedlemmar kan vara
  - konstanta
  - referenser
- men endast om de initieras med datamedlemsinitieringslistan

```
class CR_Example
{
public:
  CR_Example(int& r, int c)
    : ref {r}, con{c}
  {
    ref = 5; // x = 5
    con = 2; // fel!
  }
private:
  int& ref;
  const int con;
};

int main()
{
  int x {};
  CR_Example y {x, 1};
}
```

Konvertering med strömoperator

```
#include <sstream>

std::string to_string(T const& data)
{
  std::ostringstream oss{};
  os << data;
  return oss.str();
}

T from_string(std::string const& str)
{
  std::istringstream iss{str};
  T data;
  iss >> data;
  return data;
}
```

Namnrymd

- Funktioner och klasser kan placeras i en namnrymd.
- Namnet på namnrymden måste användas för att komma åt medlemmarna
- Används för att undvika namnkollisioner och för utbytbarhet
- `std` är ett exempel

```
namespace my
{
  //Definitioner
}
using namespace my;
using std::cout;
```

Namnrymd

```
namespace fast_math
{
  double sin(double radians);
  double cos(double radians);
}

namespace accurate_math
{
  double sin(double radians);
  double cos(double radians);
}
```

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

using namespace std;
//välj implementation
using namespace fast_math;

int main()
{
  cout << sin(0) << cos(0) << endl;
}
```

Typalias

```
using natural = unsigned long;

using suit_name = std::string;

using card == std::pair<suit, int>

using deck = std::vector<card>
```

Varför?

- Mindre att skriva
- Mer beskrivande namn
- Ett ställe att ändra på senare

Vänner

- Vändeklaration i klass
  - Kan ge en funktion skriven utanför klassen eller skriven i en annan klass tillgång till privata datamedlemmar.
  - Kan ge en annan klass tillgång till privata datamedlemmar.
- Den som skriver klassen bestämmer genom att lägga till en vändeklaration inuti klassen.
- Definition/impementation sker utanför klassen som en icke-medlem.
- **Använd inte i labbarna!** Det bryter inkapsling.

www.ida.liu.se/~TDIU20/