TDIU20

Kursupplägg och Objektorientering

Eric Ekström, Klas Arvidsson

Institutionen för datavetenskap

LINKÖPING UNIVERSITY

1 Kursinformation
2 Objektorientering
3 Undantag
4 Testning
5 Övrigt (i mån av tid)
6 Medlemsoperator
7 Fristående operator
8 Udda fåglar (och unära operatorer)

LINKÖPING UNIVERSITY

Mål

- Finns i sin helhet på kursplanen (återfinns via hemsidan)
- Objektorienterad C++
- Designa (skapa) enskilda klasser
- Designa (skapa) polymorfa klassheirarkier
- Testa implementation av klasser

Varför objektorientering

- Skapa (åter)användbara moduler
- Skapa ändringsbara moduler
- Skapa felsäkra moduler
- Främja enklare samarbete

Kursinnehåll

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

Personal

- Finns under Kontaktinformation på hemsidan
- Kursledare: Eric Ekström / Klas Arvidsson
- Examinator: Klas Arvidsson
- Kursassistent: Simon Ahrenstedt
- Assistenter:
  - Simon Ahrenstedt
  - Maria Axelsson
  - Eric Ekström
  - Nadim Lakrouz
  - Pelle Wredenborg

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 perodens 7 första veckor (15h per vecka)
  - Vad (i alla kurser) spara du till vecka 8 och tentaperioden?

Examination

- 3 Laborationer (LAB1)
  - Klockslag
  - Lista
  - Simulator
- Tentamen (DAT1)

Laborationer 3hp

- Arbeta i par (båda ansvariga för all er kod)
- Webreg för registrering och resultat
  - Anmälan senast 17/1
- Sendlab för kodinlämning
- Deadlines på hemsidan
- Bonus för redovisning och inlämning innan deadline

Datortentamen 1hp

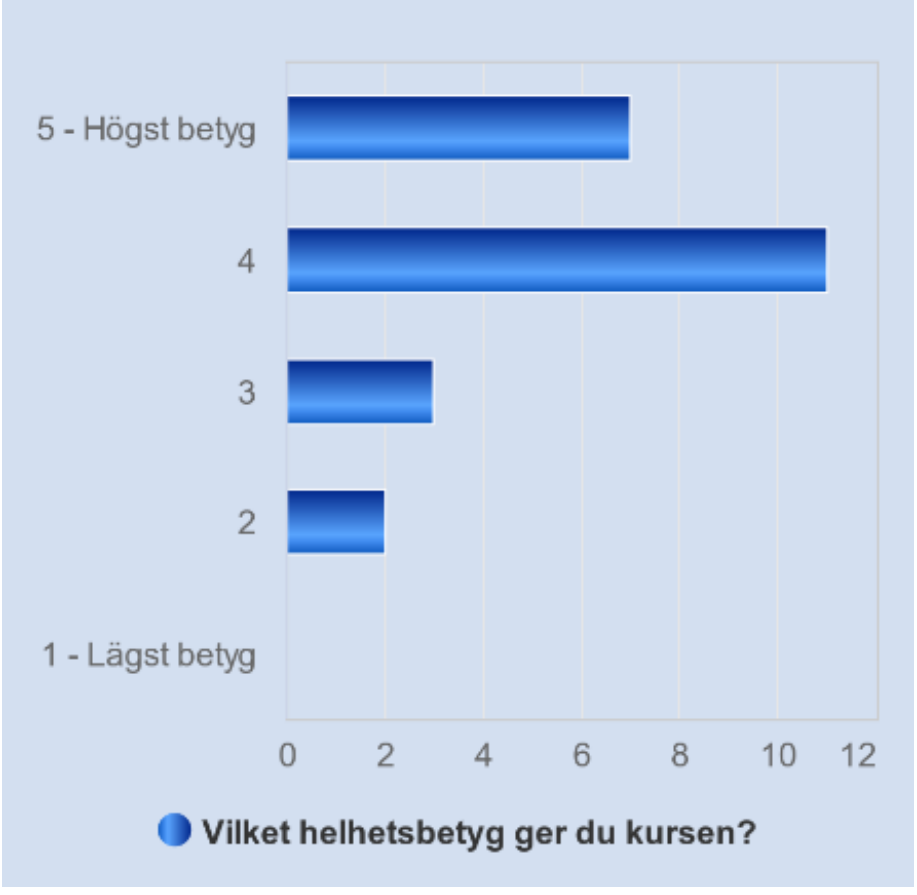
- I SU-salar
- Studentklient för frågor och inlämning
- Tentavakter för regler och ordning

- All information finns på hemsidan!

Återkoppling från tidigare år

- Lektioner mycket uppskattade.
- Labbassistenter tillgängliga och svarar gärna på frågor.
- Föreläsningar på distans tråkiga och något röriga.

- Inga ändringar gjorda jämfört med förra året.



1 Kursinformation
2 Objektorientering
3 Undantag
4 Testning
5 Övrigt (i mån av tid)
6 Medlemsoperator
7 Fristående operator
8 Udda fåglar (och unära operatorer)

LINKÖPING UNIVERSITY

Objektorientering konceptuellt

Ett sätt att beskriva för datorn hur den skall se på världen, ett sätt att skapa struktur i programvara.

Vi skapar **objekt**:

- har ansvarsområden
- samlar data och funktionalitet

Omvänt skrivsätt jämför med tidigare. Vi utgår från objektet:

Imperativt:

```
struct participant
runner{"Klas A", "UPP"};
print(runner);
give_start_no(runner, 1);
```

Objektorienterat:

```
Participant runner {"Klas A",
    "UPP"};
runner.print();
runner.give_start_no(1);
```

Skapa ett kortspel

- Vad behövs för att skapa ett kortspel?
  - Kort
  - Kortlek
- Vad ska vi kunna göra med ett kort?
  - Vi skriver kod som använder ett kort!
- Vad ska vi göra med en kortlek?
  - Vi skriver kod som använder en kortlek!

main.cc (ett förslag)

```
int main()
{
  Deck deck1{};
  deck1.shuffle();
  while( ! deck1.is_empty() )
  {
    Card card1{ deck.draw() };
    std::cout << card1.to_string() << std::endl;
  }
}
```

Skapa klasser

När vi fått ett hum om hur huvudprogrammet behöver använda kort och kortlek kan vi börja skapa klasser.

- Vilka datamedlemmar behövs?
- Vilka medlemsfunktioner behövs?
- Hur hindrar vi användaren från att göra fel?
- Hur underlättar vi för oss att göra rätt senare?

Imperativ repetition

- Funktionsöverlagring
- Standardvärde på parametrar
- Parameteröverföring
  - indata av inbyggd typ -> kopia (billigt för inbyggda datatyper)
  - indata av klasstyp -> konstant referens (undviker dyra kopior)
  - utdata -> ändringsbar referens
- returvärde
- referens

Klasser

- Inkapsling
- Konstruktor
- Medlemsfunktioner
- Ändringsskydd på medlemsfunktioner
- Skydd
- Datamedlemmar

Card.h

```
class Card
{
    public:
    Card(std::string const& s, int v);

    int get_value() const;

    bool is_red() const;

    std::string to_string() const;

    private:
    std::string suit;
    int value;
};
```

Klasser

- Konstruktor
- Datamedlems-initieringslista
- Ändringsskydd på medlemsfunktioner

- Skillnad samt likhet mellan klass, instans och objekt

Card.cc

```
Card::Card(std::string const& s, int v)
    : suit{s}, value{v}
{}

bool Card::is_red() const
{
    return (suit == "hearts" ||
            suit == "diamonds");
}
```

main.cc

```
int main()
{
    Card ace{"hearts", 1};
    Card ring{"one to rule them all", 99};
}
```

Klass

- Beskriver en datatyp sammansatt av flera variabler och tillhörande funktioner
- En variabel inuti en klass kallas datamedlem
- Klassbeskrivningen kan instansieras till objekt i programmet
  - Klass <-> ritning
- Vi kan skapa många objekt, vart och ett med sina egna värden på datamedlemmarna
  - Objekt <-> Hus (eller vad nu ritningen beskriver)

Datamedlem

- En variabel inuti en klass
- Skapas och tas bort med varje klassobjekt
- Varje klassobjekt har en egen uppsättning datamedlemmar
- Representerar klassens datainnehåll
- Beskriver relationen består av"eller "har"
- Är global inom klassen"
- Kan skyddas från åtkomst "utifrån"
- Får startvärden via konstruktors datamedlemsinitieringslista!

Medlemsfunktion

- Funktion inuti en klass
- Funktionen tillhör klassen
- Kan endast anropas utifrån ett objekt
- Har automatiskt tillgång till objektets datamedlemmar (objektet skickas med till funktionen automatiskt)

Skydd

- Innehållet i en klass kan placeras publikt (public)
  - Synligt för din kollega att anropa från alla annan programkod
- Eller privat (private)
  - Endast funktioner som redan tillhör klassen kan se och använda privata medlemmar
- Kompilatorn ser till att reglerna följs

Inkapsling

- Kollegan kan bara använda det som är publikt!
- Det är tydligt vad vi avser kollegan ska använda
- Vi kan ändra allt som är privat senare och kollegans kod påverkas inte eftersom kompilatorn förbjudit hen 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!

Konstruktor

- Funktion som måste anropas för att initiera instanser av en klass (dvs skapa objekt)
- Datamedlemsinitieringslistan initierar datamedlemmar utifrån värden på parametrar
- Inget returvärde: resultatet av ett konstruktoranrop är ett skapat objekt
- Har alltid samma namn som klassen själv
- Valfritt antal parametrar
- Defaultkonstruktor: en konstruktor utan parametrar

Ändringsskydd på medlemsfunktioner

- Anger att medlemsfunktionen inte ändrar objektet (inga datamedlemmar ändras och inga medlemsfunktioner som ändrar objektet anropas)
- Anges med const efter deklarationen
- Avsaknad av const anger ATT funktionen ändrar objektet (även om den inte gör det!)
- Förtydligar för den som anropar
- Viktigt för konstanta objekt (t.ex parametrar med const&)

1 Kursinformation
2 Objektorientering
3 Undantag
4 Testning
5 Övrigt (i mån av tid)
6 Medlemsoperator
7 Fristående operator
8 Udda fåglar (och unära operatorer)

Undantag

Vad gör vi om det blir fel under körning?

- I fösta hand, generera ett kompileringsfel.
- I andra hand, kasta ett undantag!

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

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)
```

Kasta undantag

```
Card::Card(std::string const& s, int value)
: suit{suit}, value{value}
{
    if( value < 1 || value > 13 )
    {
    throw std::logic_error{"Invalid card value"};
    }
}
```

Att kasta undantag i konstruktorn är det enda som hindrar att objektet skapas. Om du returnerar från en konstruktor kommer objektet skapas (även om du inte initierat allt)

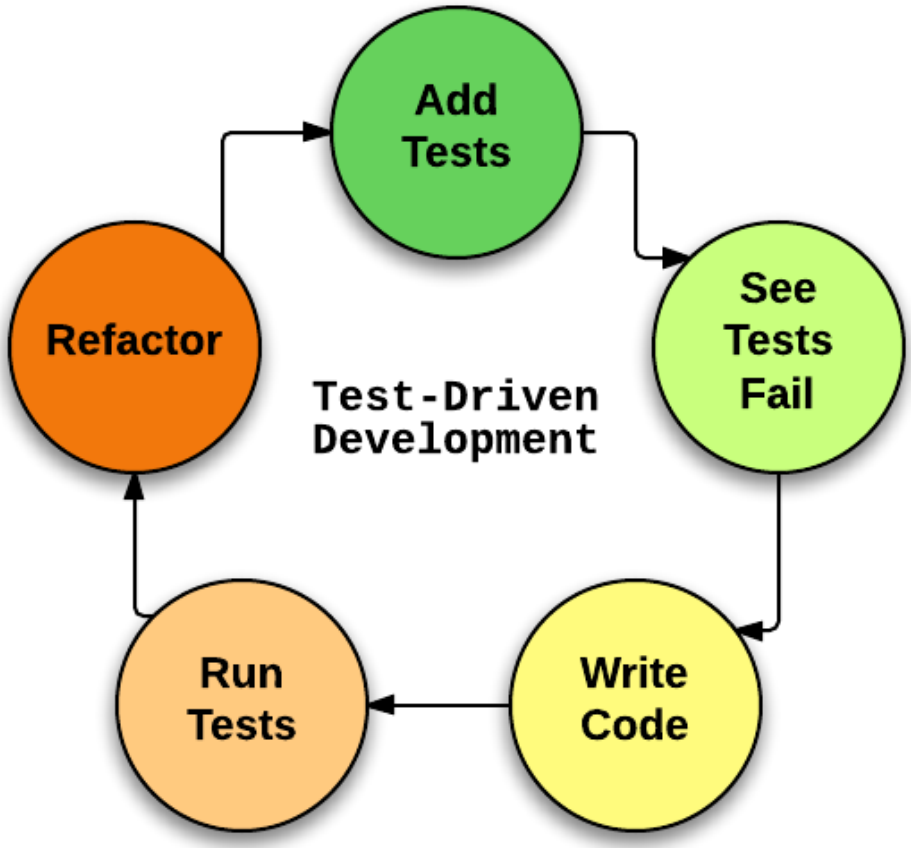
Fånga undantag

```
int main()
{
    try
    {
      Card ace{"hearts", 1};
      Card ring{"one to rule them all", 99}; //Undantag kastas här
    }
    catch(std::exception& e) //Fångar alla standard-undantag
    {
      //Skriver ut felmeddelande
      std::cout << "Error: " << e.what() << std::endl;
    }
}
```

1 Kursinformation
2 Objektorientering
3 Undantag
4 Testning
5 Övrigt (i mån av tid)
6 Medlemsoperator
7 Fristående operator
8 Udda fåglar (och unära operatorer)

LINKÖPING UNIVERSITY

Testdriven utveckling (TDD)



Catch

```
#include "catch.hpp"
#include "Card.h"

TEST_CASE( "Test constructor" ) {
  Card c {"Hearts", 1};
  CHECK( c.get_suit() == "Hearts" );
  CHECK_FALSE( c.get_value() == 0 );
}

TEST_CASE( "Test setter" ) {
  CHECK_THROWS( Card{"one to rule them all", 99} );
}
```

Filer för att testa med Catch

- `catch.hpp`
- `test_main.cc`
- Testfilen och tillhörande filer.
- Finns i givna filer för Klockslags-labben!

Snabbare kompilering

Kompilera en .o fil först, detta bör bara göras en gång

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

Därefter kan ni kompilera med följande

```
$ w++17 test_main.o person_test.cc Person.cc
```

1 Kursinformation
2 Objektorientering
3 Undantag
4 Testning
5 Övrigt (i mån av tid)
6 Medlemsoperator
7 Fristående operator
8 Udda fåglar (och unära operatorer)

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};
}
```

Konverting till och från sträng

- Från heltal till sträng:
  `to_string(num)`
- Från sträng till heltal eller flyttal:
  `stoi(str)`
  `stod(str)`
- Från enskilt tecken till sträng:
  `string(1, 'c')`

Konvertering med strömoperator

```
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

math.h

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

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

math.cc

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

using namespace std;
//choose 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 LAB! -> Det bryter inkapsling.

1 Kursinformation
2 Objektorientering
3 Undantag
4 Testning
5 Övrigt (i mån av tid)
6 Medlemsoperator
7 Fristående operator
8 Udda fåglar (och unära operatorer)

Motivation

Det är praktiskt att kunna använda de vanliga operatorerna även när operanderna är av egen klasstyp.

```
Card a{"hearts", 5};
Card b{"hearts", 9};

if ( a < b ) {}
if ( a < 7 ) {}
if ( 7 < a ) {}

std::cout << a << " " << b << std::endl;
std::cin >> a >> b;
```

Då behöver vi först specificera hur varje operator ska fungera:
https://en.cppreference.com/w/cpp/language/operators

Operander inom klassen

```
if ( card1 > card2 ) // jämförelse mellan kort
```

- Steg 1: Kompilatorn letar efter medlemsfunktionen:

```
bool Card::operator>(Card const&) const;
```

Höger operand utanför klassen

```
if ( card1 > 7 ) // jämförelse mellan kort och heltal
```

- Steg 1: Kompilatorn letar efter medlemsfunktionen:

```
bool Card::operator>(int v) const;
```

1 Kursinformation
2 Objektorientering
3 Undantag
4 Testning
5 Övrigt (i mån av tid)
6 Medlemsoperator
7 Fristående operator
8 Udda fåglar (och unära operatorer)

Vänster operand utanför klassen

```
if ( 7 > card1 ) // jämförelse mellan heltal och kort
```

- Steg 1: Kompilatorn letar efter medlemsfunktionen:

```
bool int::operator>(Card const&) const;
```

- Steg 2: När steg 1 misslyckas letar kompilatorn efter:

```
bool operator>(int, Card const&);
```

Ingen passande operator för operanderna

- Steg 3: Om steg 2 misslyckas blir det kompileringsfel!
- Hur vet du returtypen från operatorn?
- Hur vet du om du ska skriva operatorn i steg 1 eller steg 2?

Ingen passande operator för operanderna

- Hur vet du returtypen från operatorn?

=> Givet utifrån operatorn ifråga och hur du vill den ska fungera.

=> Följ konvention, standard och normal förväntan!

- Hur vet du om du ska skriva operatorn i steg 1 eller steg 2?

=> Prioritera på samma sätt som kompilatorn!

Utmatning på ström

```
std::ostream& operator<<(std::ostream& os, Card const& rhs)
{
  return os << rhs.to_string();
}
```

Inmatning från ström

```
std::istream& operator>>(std::istream& is, Card& rhs)
{
  std::string s;
  int v;
  is >> s >> v;

  try
  {
    rhs = Card{s, v};
  }
  catch(std::exception& e)
  {
    is.setstate(std::ios::failbit);
  }
  return is;
}
```

1 Kursinformation
2 Objektorientering
3 Undantag
4 Testning
5 Övrigt (i mån av tid)
6 Medlemsoperator
7 Fristående operator
8 Udda fåglar (och unära operatorer)

Pre- vs Postincrement

```
a = ++a; // preincrement
b = b++; // postincrement, return old value
```

```
Card& Card::operator++()
{
  value %= 13; // keep value in valid range: 13 --> 0 ...
  value += 1;  // ... and: 0 --> 1
  return *this;
}

Card Card::operator++(int) // extra unused int argument
{
  Card old{*this};
  ++*this; // (*this).operator++(), fine! preincrement above
  // *this++; // (*this).operator++(0), beware! recursion to self
  return old;
}
```

Indexeringsoperatorn

```
a = deck1[4]; // Indexering
```

```
Card Deck::operator[](int i)
{
  return cards.at(i);
}
```

Funktionsoperatorn

```
std::vector<Card> hand = deck1(5,7); // Funktionsanrop
```

```
std::vector<Card> Deck::operator()(int a, int b)
{
  std::vector<Card> hand;
  for (int i{a}; i < b; ++i)
    hand.push_back( cards.at(i) );

  return hand;
}
```

Typkonvertering

```
std::string s{a}; // Typkonvertering
```

```
Card::operator std::string() const
{
  return to_string();
}
```

Literaloperatorn

```
b = 5_clubs; // Literal, b = Card("clubs", 5)
b = 5_diamonds; // Literal
```

```
Card operator ""_clubs(unsigned long long v)
{
  return Card{"clubs", static_cast<int>(v)};
}

Card operator ""_diamonds(unsigned long long v)
{
  return Card{"diamonds", static_cast<int>(v)};
}
```

Piloperatorn

Vi behöver först förstå pekare, avreferering och punktoperatorn. Detta kommer senare i kursen.

Måste returnera en rå pekare eller objekt där piloperatorn kan appliceras igen.

www.liu.se