TDDC76 – Programmering och datastrukturer

Klasser, filuppdelning och operatoröverlagring

Klas Arvidsson, Eric Ekström
Institutionen för datavetenskap

LINKÖPINGS UNIVERSITET

Agenda

1. Abstraktion
2. Klasser
3. Filuppdelning
4. Inkludering och kompilering (igen)
5. Operatoröverlagring
6. Funktionsöverlagring
7. Signalera fel
8. Mer om strömmar

LINKÖPINGS UNIVERSITET

Allmän info

Arbeta hemma

- Visual Studio Code med plugin ”remote development” kan användas för att arbete hemma. Koppla upp mot antingen en lokal WSL eller mot ssh.edu.liu.se
- Tillsammans gör vi kursen bättre! Se till att vi som håller kursen får höra både vad ni uppskattar och vad ni önskar förbättra!

Dagens föreläsning

Vad ska vi göra idag?

Hur kan vi bäst skapa representationer av verkliga och abstrakta saker i våra program?

Vi ska påbörja resan in i objektorienterad programmering.

Dagens föreläsning

Vad ska vi göra idag?

- Tidigare hade vi fokuset:
  - Vad är problemet som skall lösas?
  - Vilken algoritm behövs?
  - Vilka data behöver lagras?

Dagens föreläsning

Vad ska vi göra idag?

- Tidigare hade vi fokuset:
  - Vad är problemet som skall lösas?
  - Vilken algoritm behövs?
  - Vilka data behöver lagras?

- Fokus nu, i OOP:
  - Att modellera/representera något (på ett naturligt och logiskt sätt)
  - Organisera problemet
  - Underhållsbar programkod: ändringsbarhet, felsäkerhet, återanvändbarhet

Agenda

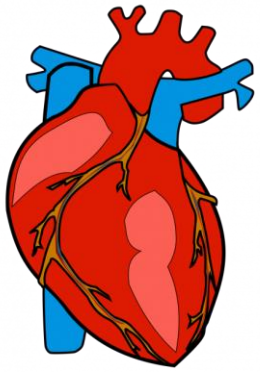
1. Abstraktion
2. Klasser
3. Filuppdelning
4. Inkludering och kompilering (igen)
5. Operatoröverlagring
6. Funktionsöverlagring
7. Signalera fel
8. Mer om strömmar

LINKÖPINGS UNIVERSITET

Abstraktion

Vad innebär abstraktion?

- Abstraktion är att skala av allt som inte är essentiellt för användning vid ett visst tillfälle
  - Ex:



Abstraktion

Vad innebär abstraktion?

- Abstraktion är att skala av allt som inte är essentiellt för användning vid ett visst tillfälle
  - Behöver vi känna till alla detaljer?

Abstraktion

Vad innebär abstraktion?

- Abstraktion är att skala av allt som inte är essentiellt för användning vid ett visst tillfälle
  - Gör något enkelt att använda trots att vi inte vet alla detaljer hur det fungerar
  - En bra abstraktion gör även svårt att använda på fel sätt (säkert att använda)
- Vilka sätt att skapa abstraktioner känner du till?
- Vilka abstraktioner har du använt?
- Är C++ en abstraktion?

Abstraktion

Vad innebär abstraktion?

- Abstraktion är en grundbult i OOP
  - Hantera komplexitet genom att undvika onödiga detaljer för användaren
  - Ger användaren möjlighet att implementera komplexa program ovanpå abstraktionen – utan att förstå eller behöva tänka på dess underliggande komplexitet

Agenda

1. Abstraktion
2. Klasser
3. Filuppdelning
4. Inkludering och kompilering (igen)
5. Operatoröverlagring
6. Funktionsöverlagring
7. Signalera fel
8. Mer om strömmar

LINKÖPINGS UNIVERSITET

C++ är ett språk med klass

Klass: En sammansatt datatyp

- Är en sammansatt datatyp
  - Vi kan samla attribut och funktioner i en klass
- Eftersom det är en datatyp kan vi skapa variabler av den
  - En variabel av en klass kallas ett objekt/en instans
- Vi kan anropa funktioner som tillhör objektet

C++ är ett språk med klass

Klass: grundläggande skillnader i tanke och skrivsätt

```
// Imperativ "C"
struct Person
{
  …
};

double jump(Person const& p)
{
  return p.shoe_size + p.length;
}

int main()
{
  Person klas{"Klas", 42, 1.89};
  jump(klas);
}
```

```
// Objektorienterad C++
class Person
{
  double jump() const;
  …
};

double Person::jump() const
{
  return shoe_size + length;
}

int main()
{
  Person klas{"Klas", 42, 1.89};
  klas.jump();
}
```

Sammansatta datatyper

Fallstudie: Kaffekokare

- Har attribut
  - Information/egenskaper
- Har funktioner
  - Vad den kan göra, ”Operationer”
- Attribut och funktioner hänger ihop för att bilda objektet
  - Sammansatt datatyp

Objekt:

Sammansatta datatyper

Fallstudie: Kaffekokare

- Vill kunna använda objektet (med dess attribut och funktioner) utan att behöva tänka på dess underliggande implementation
- För att skapa bra abstraktion lönar det sig ofta att börja med att tänka ut hur vi vill använda den (top-down)

Objekt:

Sammansatta datatyper

Fallstudie: Kaffekokare

coffe_program.cc

```
#include <iostream>
#include "coffee_machine.h"
using namespace std;
int main()
{
   Coffee_Machine brew_master{};
   brew_master.fill_water(4.0);
   brew_master.fill_coffee("Zoega Hazienda", 4.0);
   brew_master.brew_coffee();

   if (brew_master.is_done())
   {
      cout << brew_master.get_cups() << " cups of "
           << brew_master.get_coffee_name()
           << " is done!" << endl;
   }
   return 0;
}
```

Objekt:

Sammansatta datatyper

Fallstudie: Kaffekokare

coffe_program.cc

```
#include <iostream>
#include "coffee_machine.h"
using namespace std;
int main()
{
   Coffee_Machine brew_master{};
   brew_master.fill_water(4.0);
   brew_master.fill_coffee("Zoega Hazienda", 4.0);
   brew_master.brew_coffee();

   if (brew_master.is_done())
   {
      cout << brew_master.get_cups() << " cups of "
           << brew_master.get_coffee_name()
           << " is done!" << endl;
   }
   return 0;
}
```

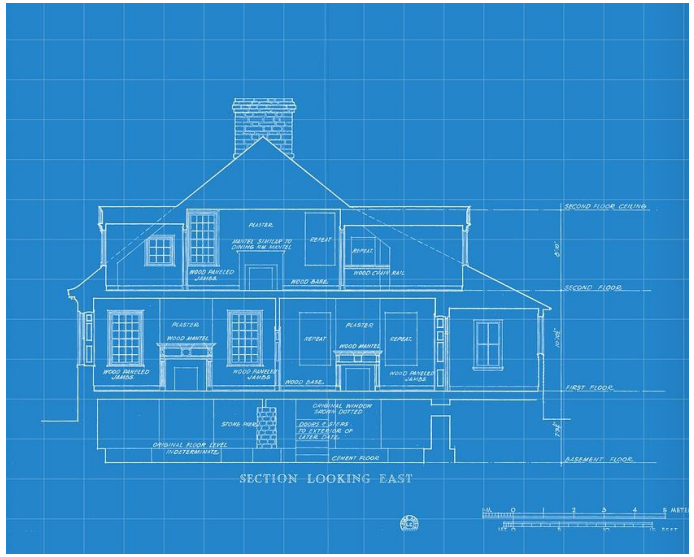
- Top-Down metod
- Nu vet vi användningen
- Hur skriver vi koden som möjliggör detta?

Klasser

Skillnad mellan klass och instans

- Klass är ritningen, en instans är vad vi skapar från ritningen

Klass:



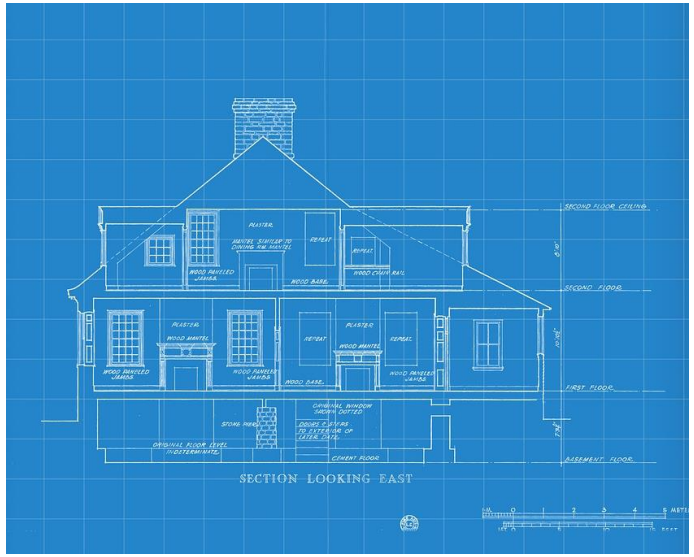
Instans (objekt):

Klasser

Skillnad mellan klass och instans

- Av en klass kan vi skapa många olika instanser, som alla har olika attribut (egenskaper)

Klass:



Instanser (objekt):

Klasser

Vi har arbetat med klasser innan!

- Klasser vi har arbetat med tidigare: string, istream, ostream
  - cin är en instans av klassen istream
  - cout är en instans av klassen ostream
- De har också tillhörande funktioner:
  - str.length(), str.at(2)
  - cin.ignore(…), cin.fail()

Klasser

Vi har arbetat med klasser innan!

- Klasser vi har arbetat med tidigare: string, istream, ostream
  - cin är en instans av klassen istream
  - cout är en instans av klassen ostream
- De har också tillhörande funktioner:
  - str.length(), str.at(2)
  - cin.ignore(…), cin.fail()
- Det har gått bra utan att vi känt till hur de är implementerade!

Klasser

Vi har arbetat med klasser innan!

- Klasser vi har arbetat med tidigare: string, istream, ostream
  - cin är en instans av klassen istream
  - cout är en instans av klassen ostream
- De har också tillhörande funktioner:
  - str.length(), str.at(2)
  - cin.ignore(...), cin.fail()
- Det har gått bra utan att vi känt till hur de är implementerade!
- (Tips: går alltid att läsa mer om klasserna på cppreference.com. Punkt-operatorn (instans.funktion) avslöjar att vi har med en instans av en klass att göra.)

Klasser

Kaffekokare

coffe_program.cc

```
#include <iostream>
#include "coffee_machine.h"
using namespace std;
int main()
{
   Coffee_Machine brew_master{};
   brew_master.fill_water(4.0);
   brew_master.fill_coffee("Zoega Hazienda", 4.0);
   brew_master.brew_coffee();

   if (brew_master.is_done())
   {
      cout << brew_master.get_cups() << " cups of "
           << brew_master.get_coffee_name()
           << " is done!" << endl;
   }
   return 0;
}
```

- Nu vet vi användningen
- Hur skriver vi koden som möjliggör detta?

Klasser

Klassdefinition

```
#include <string>

class Coffee_Machine
{
public:
   Coffee_Machine(); //Konstruktor

   void fill_water(double cups);
   void fill_coffee(std::string const & name,
                    double scoops);
   void brew_coffee();
   bool is_done() const;

private:
   double water_amount{};
   double coffee_amount{};
   std::string coffee_name{};

};
```

- Definition av klassen Coffe_Machine
- Innehåller deklarationer av
  - medlemsfunktioner
  - datamedlemmar (attribut)

Klasser

Klassdefinition

```
#include <string>

class Coffee_Machine
{
public:
   Coffee_Machine(); //Konstruktor

   void fill_water(double cups);
   void fill_coffee(std::string const & name,
                    double scoops);
   void brew_coffee();
   bool is_done() const;

private:
   double water_amount{};
   double coffee_amount{};
   std::string coffee_name{};

};
```

- Definition av klassen Coffe_Machine
- Innehåller deklarationer av
  - **medlemsfunktioner**
  - datamedlemmar (attribut)

Klasser

Klassdefinition

```
#include <string>

class Coffee_Machine
{
public:
   Coffee_Machine(); //Konstruktor

   void fill_water(double cups);
   void fill_coffee(std::string const & name,
                    double scoops);
   void brew_coffee();
   bool is_done() const;

private:
   double water_amount{};
   double coffee_amount{};
   std::string coffee_name{};

};
```

- Definition av klassen Coffe_Machine
- Innehåller deklarationer av
  - medlemsfunktioner
  - **datamedlemmar (attribut)**

Klasser

Synlighet

```
#include <string>

class Coffee_Machine
{
public:
   Coffee_Machine(); //Konstruktor

   void fill_water(double cups);
   void fill_coffee(std::string const & name,
                    double scoops);
   void brew_coffee();
   bool is_done() const;

private:
   double water_amount{};
   double coffee_amount{};
   std::string coffee_name{};

};
```

- I en klass kan vi bestämma synlighet på dess funktioner och variabler
- Allt under public är möjligt att använda ”utifrån”, för de som skapar en instans av vår klass
- Allt under private går inte att använda ”utifrån”, för de som skapar en instans av vår klass

Klasser

Synlighet

```
#include <string>

class Coffee_Machine
{
public:
   Coffee_Machine(); //Konstruktor

   void fill_water(double cups);
   void fill_coffee(std::string const & name,
                    double scoops);
   void brew_coffee();
   bool is_done() const;

private:
   double water_amount{};
   double coffee_amount{};
   std::string coffee_name{};

};
```

- I en klass kan vi bestämma synlighet på dess funktioner och variabler
- Allt under **public** är möjligt att använda ”utifrån”, för de som skapar en instans av vår klass
- Allt under private går inte att använda ”utifrån”, för de som skapar en instans av vår klass

Klasser

Synlighet

```
#include <string>

class Coffee_Machine
{
public:
   Coffee_Machine(); //Konstruktor

   void fill_water(double cups);
   void fill_coffee(std::string const & name,
                    double scoops);
   void brew_coffee();
   bool is_done() const;

private:
   double water_amount{};
   double coffee_amount{};
   std::string coffee_name{};

};
```

- I en klass kan vi bestämma synlighet på dess funktioner och variabler
- Allt under public är möjligt att använda ”utifrån”, för de som skapar en instans av vår klass
- Allt under **private** går inte att använda ”utifrån”, för de som skapar en instans av vår klass.
  - Den egna klassen kan komma åt allt under private.

Klasser

Synlighet

```
#include <string>

class Coffee_Machine
{
public:
   Coffee_Machine(); //Konstruktor

   void fill_water(double cups);
   void fill_coffee(std::string const & name,
                    double scoops);
   void brew_coffee();
   bool is_done() const;

private:
   double water_amount{};
   double coffee_amount{};
   std::string coffee_name{};

};
```

- Allt under **private** går inte att använda ”utanför”, för de som skapar en instans av vår klass

- Ex:

coffe_program.cc

```
#include <iostream>
#include ”coffee_machine.h”
using namespace std;

int main()
{
   Coffee_Machine brew_master{};
   brew_master.water_amount = 20.0;

   return 0;
}
```

Klasser

Synlighet

```
#include <string>

class Coffee_Machine
{
public:
   Coffee_Machine(); //Konstruktor

   void fill_water(double cups);
   void fill_coffee(std::string const & name,
                    double scoops);
   void brew_coffee();
   bool is_done() const;

private:
   double water_amount{};
   double coffee_amount{};
   std::string coffee_name{};

};
```

- Allt under **private** går inte att använda ”utanför”, för de som skapar en instans av vår klass
- Ex:

coffe_program.cc

```
#include <iostream>
#include ”coffee_machine.h”
using namespace std;

int main()
{
   Coffee_Machine brew_master{};
   brew_master.water_amount = 20.0;

   return 0;
}
```

KOMPILERAR INTE

Klasser

Synlighet

```
#include <string>

class Coffee_Machine
{
public:
   Coffee_Machine(); //Konstruktor

   void fill_water(double cups);
   void fill_coffee(std::string const & name,
                    double scoops);
   void brew_coffee();
   bool is_done() const;

private:
   double water_amount{};
   double coffee_amount{};
   std::string coffee_name{};

};
```

- I en klass kan vi bestämma synlighet på dess funktioner och variabler
- public
- private
- **protected** – kommer nästa vecka

Klasser

Synlighet

- Tänk tillbaka på liknelsen
  - Klass – husritning
  - Instans – ett visst hus enligt husritningen

- De delar av ritningen som är *public* känner *alla* till (instanser och utomstående)
  - Ex: Fasad, fönster, ytterdörr, uteplats

- De delar av ritningen som är *private* känner bara invigda till (*alla instanser*)
  - Ex: Var toaletten är placerad eller “allt inuti huset”

Klasser

Synlighet

- Tänk tillbaka på liknelsen
  - Klass – husritning
  - Instans – ett visst hus enligt husritningen

- De delar av ritningen som är *public* kan *alla* använda
  - (Fasad, fönster, ytterdörr, uteplats)
  - Måla(), putsa(), knacka(), grilla()

- Det som är *private* kan bara invigda använda
  - (Toalett, kök, vardagsrum, sovrum, …)
  - Klossa(), laga mat(), inreda(), …()

Klasser

Synlighet – varför använda private?

- Ändringsbarhet: endast klassens funktioner kan använda det som är private, så senare ändringar kan hållas lokalt i klassen

- Felsäkerhet: endast klassens funktioner kan använda det som är private, så vi har god kontroll över vilka värden som kan förekomma och varifrån fel kan uppstå

- Tydlighet: klar signal till andra programmerar att ”det här ska du inte använda eller tänka på, det är privat för klassen”

Klasser

Konstruktorn

```
#include <string>

class Coffee_Machine
{
public:
   Coffee_Machine(); //Konstruktor

   void fill_water(double cups);
   void fill_coffee(std::string const & name,
                    double scoops);
   void brew_coffee();
   bool is_done() const;

private:
   double water_amount{};
   double coffee_amount{};
   std::string coffee_name{};

};
```

- **Konstruktorn** är en speciell medlemsfunktion
- När vi initialiserar en instans av klassen

```
Coffee_Machine brew_master{};
```

  kommer konstruktorn att köras.
- Tanken är att vi där i tilldelar värden till de datamedlemmar som behövs när objektet ”föds”

Klasser

Konstruktorn

```
#include <string>

class Coffee_Machine
{
public:
   Coffee_Machine(); //Konstruktor

   void fill_water(double cups);
   void fill_coffee(std::string const & name,
                    double scoops);
   void brew_coffee();
   bool is_done() const;

private:
   double water_amount{};
   double coffee_amount{};
   std::string coffee_name{};

};
```

- **Deklaration** av konstruktorn i klassdefintionen
- **Definitionen** av konstruktorn ser ut som följande:

```
Coffee_Machine::Coffee_Machine()
: water_amount{0}, coffee_amount{0},
  coffee_name{""}
{
   //Här kan vi skriva instruktioner
}
```

Klasser

Konstruktorn

```
#include <string>

class Coffee_Machine
{
public:
   Coffee_Machine(); //Konstruktor

   void fill_water(double cups);
   void fill_coffee(std::string const & name,
                    double scoops);
   void brew_coffee();
   bool is_done() const;

private:
   double water_amount{};
   double coffee_amount{};
   std::string coffee_name{};

};
```

- Deklaration av konstruktorn i klassdefintionen
- Definitionen av konstruktorn ser ut som följande:

```
Coffee_Machine::Coffee_Machine()
: water_amount{0}, coffee_amount{0},
  coffee_name{""}
{
   //Här kan vi skriva instruktioner
}
```

- Efter : följer **datamedlemsinitialiseringslistan** här får datamedlemmarna sina start-värden

Klasser

Initialiseringslistan

- Varför behöver vi ge variablerna sina start-värden i datamedlemsinitialiseringslistan?
- Vad är problemen att initialisera efteråt?

```
int main()
{
   Coffee_Machine brew_master{};
   brew_master.water_amount = 0.0;
   brew_master.coffee_amout = 0.0;
   brew_master.coffee_name  = "";
   return 0;
}
```

  - Exponerar interna detaljer!
  - Ingen felkontroll!
  - Opraktiskt!
  - Vi vill se klassen som en inkapslad enhet som är enkel och säker att använda

Klasser

Flera konstruktorer

```
#include <string>

class Coffee_Machine
{
public:
   Coffee_Machine();
   Coffee_Machine(double water, double coffee
                  std::string name);

   void fill_water(double cups);
   void fill_coffee(std::string const & name,
                    double scoops);
   void brew_coffee();
   bool is_done() const;

private:
   double water_amount{};
   double coffee_amount{};
   std::string coffee_name{};

};
```

- Vi kan ha flera konstruktorer
  - Säg att vi ska kunna skapa en instans som har vatten, kaffe och kaffesort från start
- Definitionen av andra konstruktorn:

```
Coffee_Machine::Coffee_Machine(double water,
            double coffee, std::string name)
: water_amount{water},
  coffee_amount{coffee},
  coffee_name{name}
{
   //Här kan vi skriva instruktioner
}
```

Klasser

Flera konstruktorer

```
#include <string>

class Coffee_Machine
{
public:
   Coffee_Machine();
   Coffee_Machine(double water, double coffee
                  std::string name);

   void fill_water(double cups);
   void fill_coffee(std::string const & name,
                    double scoops);
   void brew_coffee();
   bool is_done() const;

private:
   double water_amount{};
   double coffee_amount{};
   std::string coffee_name{};

};
```

- Hur vet programmet vilken konstruktor den ska köra?

Klasser

Flera konstruktorer

```
#include <string>

class Coffee_Machine
{
public:
   Coffee_Machine();
   Coffee_Machine(double water, double coffee
                  std::string name);

   void fill_water(double cups);
   void fill_coffee(std::string const & name,
                    double scoops);
   void brew_coffee();
   bool is_done() const;

private:
   double water_amount{};
   double coffee_amount{};
   std::string coffee_name{};

};
```

- Hur vet programmet vilken konstruktor den ska köra?

```
int main()
{
  Coffee_Machine brew_master{};

  Coffee_Machine brew_master{4.0,
                             4.0,
                             "Gevalia"};

  return 0;
}
```

Klasser

Flera konstruktorer

```
#include <string>

class Coffee_Machine
{
public:
   Coffee_Machine();
   Coffee_Machine(double water, double coffee
                  std::string name);

   void fill_water(double cups);
   void fill_coffee(std::string const & name,
                    double scoops);
   void brew_coffee();
   bool is_done() const;

private:
   double water_amount{};
   double coffee_amount{};
   std::string coffee_name{};

};
```

- Hur vet programmet vilken konstruktor den ska köra?

```
int main()
{
  Coffee_Machine brew_master{};

  Coffee_Machine brew_master{4.0,
                             4.0,
                             "Gevalia"};

  return 0;
}
```

Klasser

Kan klassen vara utan konstruktor?

```
#include <string>

class Coffee_Machine
{
public:
   //Här skulle vi ha deklarerat en konstruktor

   void fill_water(double cups);
   void fill_coffee(std::string const & name,
                    double scoops);
   void brew_coffee();
   bool is_done() const;

private:
   double water_amount{};
   double coffee_amount{};
   std::string coffee_name{};

};
```

- Klassen kommer aldrig vara utan konstruktor
- Om vi själva inte implementerar en konstruktor så skapas det en automatiskt - default konstruktorn
- Default konstruktorn är helt tom på innehåll

Klasser

Kan programmet ha ingen konstruktor?

```
#include <string>

class Coffee_Machine
{
public:
   Coffee_Machine() = default;

   void fill_water(double cups);
   void fill_coffee(std::string const & name,
                    double scoops);
   void brew_coffee();
   bool is_done() const;

private:
   double water_amount{};
   double coffee_amount{};
   std::string coffee_name{};

};
```

- Nej! Om vi själva inte implementerar en konstruktor så skapas det en automatiskt - default konstruktorn
- Default konstruktorn är helt tom på innehåll
- Vi kan explicit skriva default konstruktorn för tydlighet

Klasser

Kaffekokare – Medlemsfunktioner som inte ändrar datamedlemmar

```
#include <string>

class Coffee_Machine
{
public:
   Coffee_Machine(); //Konstruktor

   void fill_water(double cups);
   void fill_coffee(std::string const & name,
                    double scoops);
   void brew_coffee();
   bool is_done() const;

private:
   double water_amount{};
   double coffee_amount{};
   std::string coffee_name{};

};
```

- I vissa medlemsfunktioner vill vi inte ändra på någon datamedlem
  - Detta signaleras genom att sätta funktionen till const

Klasser

Kaffekokare – Medlemsfunktioner som inte ändrar datamedlemmar

```
#include <string>

class Coffee_Machine
{
public:
   Coffee_Machine(); //Konstruktor

   void fill_water(double cups);
   void fill_coffee(std::string const & name,
                    double scoops);
   void brew_coffee();
   bool is_done() const;

private:
   double water_amount{};
   double coffee_amount{};
   std::string coffee_name{};

};
```

- I vissa medlemsfunktioner vill vi inte ändra på någon datamedlem
  - Detta signaleras genom att sätta funktionen till const
- **is_done** ska bara kontrollera om maskinen är klar (allt vatten är förbrukat)

```
bool Coffee_Machine::is_done() const
{
    if (water_amount == 0.0)
    {
       return true;
    }
    return false;
}
```

Klasser

Kaffekokare – Medlemsfunktioner som inte ändrar datamedlemmar

```
#include <string>

class Coffee_Machine
{
public:
   Coffee_Machine(); //Konstruktor

   void fill_water(double cups);
   void fill_coffee(std::string const & name,
                    double scoops);
   void brew_coffee();
   bool is_done() const;

private:
   double water_amount{};
   double coffee_amount{};
   std::string coffee_name{};

};
```

- const medlemsfunktioner

```
bool Coffee_Machine::is_done() const
{
    if (water_amount == 0)
    {
       return true;
    }
    water_amount = 0; // fel, vi lovade
    return false;
}
```

VARFÖR KOMPILERAR DET INTE?

```
int main()
{
  const Coffee_Machine master{4.0,
                              4.0,
                              "Gevalia"};
  master.brew_coffee(); // fel, den lovar inte
  master.is_done();
  return 0;
}
```

Klasser

Var skriver vi koden?

- Vi har lärt oss hur vi skriver:
  - klassdefinition med deklarationer av medlems-funktioner och -variabler
  - Definitioner av konstruktorer och medlemsfunktioner
- I vilka filer ska vi skriva dessa definitioner och deklarationer?

Paus!

Kan du begreppen?

Definition?
Deklaration?

Konstruktor?
Datamedlems-initieringslista?
Const-funktion?

Klass?
Instans?
Objekt?

Abstraktion?

Agenda

1. Abstraktion
2. Klasser
3. Filuppdelning
4. Inkludering och kompilering (igen)
5. Operatoröverlagring
6. Funktionsöverlagring
7. Signalera fel
8. Mer om strömmar

LINKÖPINGS UNIVERSITET

Klasser

Var skriver vi koden?

- Vi har lärt oss hur vi skriver:
  - klassdefinition med deklarationer av medlems-funktioner och -variabler
  - Definitioner av konstruktorer och medlemsfunktioner
- I vilka filer ska vi skriva dessa defintioner och deklarationer?

Klasser

Var skriver vi koden?

- Vi vill kunna
  - Låta andra program använda våra klasser
  - Återanvända våra klasser
    (Dvs. vi ska kunna vara lata senare)

- Därför behöver vi skriva klassen så det blir enkelt att importera bara den senare.

```
class A_Class {
public:
  void func() {
     a_variable = 10;
  }

private:
  int a_variable{};
};

int main(){
   A_Class a{};
   a.func();
   return 0;
}
```

Allt i samma fil:
Inte återanvändbart

Klasser

Var skriver vi koden?

- Klassdefinitioner ligger i en .h-fil (header-fil)

a_class.h

```
class A_Class {
public:
  A_Class();
  void func();
private:
  int a_variable{};
};
```

Klasser

Var skriver vi koden?

- Klassdefinitioner ligger i en .h-fil (header-fil)

a_class.h

```
class A_Class {
public:
  A_Class();
  void func();
private:
  int a_variable{};
};
```

- Funktionsdefinitioner ligger i en .cc-fil

a_class.cc

```
#include "a_class.h"

A_Class::A_Class()
: a_variable{0}
{
}

void A_Class::func()
{
  a_variable = 10;
}
```

Klasser

Var skriver vi koden?

- Andra program inkluderar h-filen och kompilerar cc-filen
- Klassdefinitioner ligger i en .h-fil (header-fil)
- Funktionsdefinitionern a ligger i en .cc-fil

main.cc

```
#include "a_class.h"
int main()
{
  A_Class a{};
  a.func();
  return 0;
}
```

a_class.h

```
class A_Class {
public:
  A_Class();
  void func();
private:
  int a_variable{};
};
```

a_class.cc

```
#include "a_class.h"

A_Class::A_Class()
: a_variable{0}
{
}

void A_Class::func()
{
  a_variable = 10;
}
```

Klasser

Varför har vi en .h-fil?

- .h-filen blir ett gränssnitt, ”en mellanhand”, som ett program kan vända sig till för få använda sig av klassens funktioner och variabler (om synliga)
- Funktionsdefinitionerna i .cc-filen är gömda för den som använder klassen
- Tänk en meny på en restaurang:
  - Vi kan bara beställa det som finns på menyn (*gränssnitt*, *deklarationer*)
  - Vi får inte se hur de tillagar maten i köket (*definitioner*)
  - Råvaror och funktioner som köket använder sig av kommer vi inte åt (de är *private*)

Användande program ⟷ .h-fil ⟷ .cc-fil

Kund ⟷ Meny ⟷ Köket

Agenda

1. Abstraktion
2. Klasser
3. Filuppdelning
4. Inkludering och kompilering (igen)
5. Operatoröverlagring
6. Funktionsöverlagring
7. Signalera fel
8. Mer om strömmar

LINKÖPINGS UNIVERSITET

Klasser

Kompilering – ”undefined reference to”

- Hur får anropande programmet tillgång till definitionerna?

main.cc

```
#include ”a_class.h”
int main()
{
  A_Class a{};
  a.func();
  return 0;
}
```

a_class.h

```
class A_Class {
public:
  A_Class();
  void func();
private:
  int a_variable{};
};
```

a_class.cc

```
#include ”a_class.h”

A_Class::A_Class()
: a_variable{0}
{
}

void A_Class::func()
{
  a_variable = 10;
}
```

- När du kompilerar enbart main.cc

g++ -std=c++17 main.cc

- Ger ”undefined reference to A_class::func()”.

Klasser

Kompilering

- Hur får anropande programmet tillgång till definitionerna?

main.cc

```
#include "a_class.h"
int main()
{
  A_Class a{};
  a.func();
  return 0;
}
```

a_class.h

```
class A_Class {
public:
  A_Class();
  void func();
private:
  int a_variable{};
};
```

a_class.cc

```
#include "a_class.h"

A_Class::A_Class()
: a_variable{0}
{
}

void A_Class::func()
{
  a_variable = 10;
}
```

- Kompilerar main.cc och a_class.cc tillsammans

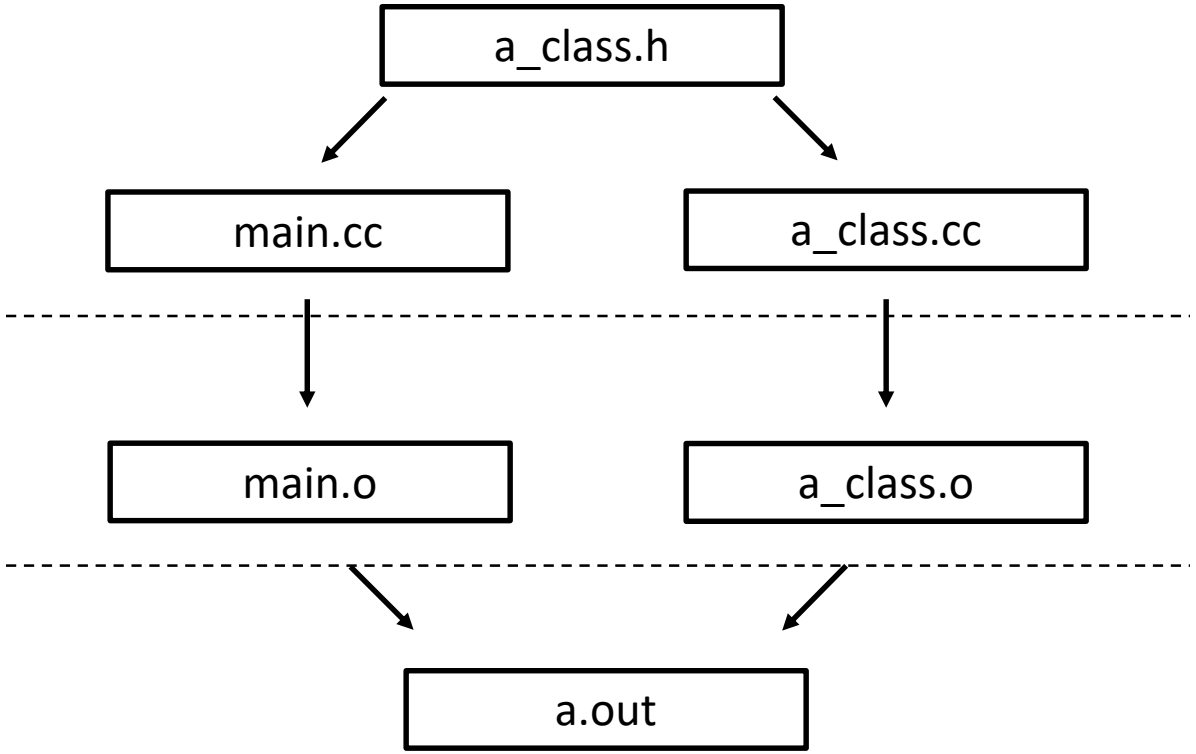
g++ -std=c++17 main.cc **a_class.cc**

Kompilera endast cc-filer!

- Två filer kompileras -> ger en gemensam körbar fil (a.out)

Klasser

Kompilering (BILD)



Steg:

- Pre-processor
  (inkludering)
- Kompilering
  (skapar kompilerade .o-filer)
- Länkning
  (länkar samman .o-filer till en körbar fil)

Klasser

Kompilering

- Alla .cc filer *kompileras* och alla .h filer *inkluderas*
- Kompilera ***aldrig*** .h-filer! Det ger problem:
  - En .gch-fil genereras. Om ni gör ändringar i .h–filen så kommer detta inte uppdateras i .gch-filen
  - När en annan fil försöker inkludera en .h-fil så kollar den först om det finns en .gch-fil och om det stämmer så inkluderas .gch-filen

Klasser

Inkludering av h-fil

- #include ”filnamn.h”: Koden i .h-filen kopieras in där inkluderingen sker (precis som när vi inkluderat tidigare)
  - ”filnamn” innebär att preprocessorn letar efter filnamn i din lokala kodmapp
  - <filnamn> innebär att preprocessorn letar efter filnamn i systemets standardmappar

main.cc

```
#include ”a_class.h”
int main()
{
  A_Class a{};
  a.func();
  return 0;
}
```

a_class.h

```
class A_Class {
public:
  A_Class();
  void func();
private:
  int a_variable{};
}
```

a_class.cc

```
#include ”a_class.h”

A_Class::A_Class()
: a_variable{0}
{
}

void A_Class::func()
{
  a_variable = 10;
}
```

Klasser

Inkludering av h-fil

- Föreställ att ett program inkluderar en .h-fil och ett annat program som i sin tur inkluderar samma .h-fil:

main.cc

```
#include "a_class.h"
#include "b_class.h"

int main()
{
…
```

a_class.h

```
#include "b_class.h"

class A_Class
{
…
```

b_class.h

```
class B_Class
{
…
```

Klasser

Inkludering av h-fil

- Föreställ att ett program inkluderar en .h-fil och ett annat program som i sin tur inkluderar samma .h-fil:

main.cc

```
#include "b_class.h"

class A_Class
{
…
```

```
#include "b_class.h"

int main()
{
…
```

b_class.h

```
class B_Class
{
…
```

Klasser

Inkludering av h-fil

- Föreställ att ett program inkluderar en .h-fil och ett annat program som i sin tur inkluderar samma .h-fil:

main.cc

```
class B_Class
{
…

class A_Class
{
…

class B_Class
{
…

int main()
{
…
```

Klasser

Inkludering av h-fil

- Föreställ att ett program inkluderar en .h-fil och ett annat program som i sin tur inkluderar samma .h-fil:
- Dubbel definition!

main.cc

```
class B_Class
 {
 …

class A_Class
{
…

class B_Class
{
…

int main()
{
…
```

KOMPILERAR INTE

Klasser

Inkludering av h-fil

- Föreställ att ett program inkluderar en .h-fil och ett annat program som i sin tur inkluderar samma .h-fil:

- Lös det med en ***header-guard*** i varje h-fil!

main.cc

```
#include "a_class.h"
#include "b_class.h"

int main()
{
…
```

a_class.h

```
#include "b_class.h"

class A_Class
{
…
```

b_class.h

```
#ifndef B_CLASS_H
#define B_CLASS_H
class B_Class
{
};
#endif
```

Klasser

Inkludering av h-fil

- Header guard ska ***alltid*** finnas i h-filer. Testa den genom att inkludera din h-fil två gånger. Det ska fungera.
- Eftersom koden i .h-filen kopieras in där den inkluderas är det viktigt att vi aldrig har `using namespace std;` i .h-filen
- Har vi `using namespace std` i .h-filen så tvingar vi på ett namespace till alla som inkluderar vår fil
  - Detta kan skapa krockar ifall de exempelvis skrivit en egen klass för strängar (istf std::string). Då vet inte programmet vilken som menas - den i iostream eller den egenskrivna.

Agenda

1. Abstraktion
2. Klasser
3. Filuppdelning
4. Inkludering och kompilering (igen)
5. Operatoröverlagring
6. Funktionsöverlagring
7. Signalera fel
8. Mer om strömmar

LINKÖPINGS UNIVERSITET

Operatoröverlagring

Att använda operatorer på sina klasser

- Säg att vi har en klass som representerar datum

date.h

```
#ifndef CLASS_H
#define CLASS_H

class Date
{
public:
  Date(int y, int m, int d);

private:
  int year{};
  int month{};
  int day{};
};

#endif
```

Operatoröverlagring

Att använda operatorer på sina klasser

- Säg att vi har en klass som representerar datum

date.h

```
#ifndef CLASS_H
#define CLASS_H

class Date
{
public:
  Date(int y, int m, int d);

private:
  int year{};
  int month{};
  int day{};
};

#endif
```

main.cc

```
int main()
{
  Date d1{2019,9,9}
  Date d2{1992,2,27}
  int days_past{};

  days_past = d1 - d2;
  cout << d1 << d2 << endl;
  return 0;
}
```

Vi vill använda operatorer på klassen, precis som vanligt!

Operatoröverlagring

Att använda operatorer på sina klasser

- Säg att vi har en klass som representerar datum

date.h

```
#ifndef CLASS_H
#define CLASS_H

class Date
{
public:
  Date(int y, int m, int d);

private:
  int year{};
  int month{};
  int day{};
};

#endif
```

main.cc

```
int main()
{
  vector<Date> v{
    Date{2019,9, 9},
    Date{1992,2,27},
    ...
  };
  for ( i = ... )
    for ( j = ... )
      if (v.at(i) > v.at(j))
        swap(v.at(i),v.at(j));
  return 0;
}
```

Vi vill använda operatorer på klassen, precis som vanligt!

Operatoröverlagring

Hur skriver vi en operator? Två alternativ

Alternativ #1: medlemsfunktion, *left operand* utgörs av objektet framför anropet

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

Alternativ #2: utomstående (fri) funktion, dvs ej medlem av någon klass

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

Operatoröverlagring

Exempel (alternativ 1)

date.cc

```
int Date::operator-(Date const & rhs)
{
  //Operatorn anropas på det vänstra objektet
  return [beräkning av skillnaden mellan lhs och rhs];
}
```

main.cc

```
int main()
{
  Date d1{2019,9,9);
  Date d2{1992,2,27};
  int days_past{};
  days_past = d1 - d2; // föreställ dig anropet d1.operator-(d2)
  return 0;
}
```

Operatoröverlagring

Exempel (alternativ 2)

date.cc

```
int operator-(Date const & lhs, Date const & rhs)
{
  return [beräkning av skillnaden mellan lhs och rhs];
}
```

main.cc

```
int main()
{
  Date d1{2019,9,9);
  Date d2{1992,2,27};
  int days_past{};
  days_past = d1 - d2; // föreställ dig anropet operator-(d1, d2)
  return 0;
}
```

Operatoröverlagring

En eller två parametrar?

- Vilken av alternativen ska vi använda när?
- Det är okej att använda vilken ni vill. Det finns däremot stunder vi blir tvingade att använda alternativ #2 (att ta in två parametrar)

main.cc

```
int main()
{
  Date d1{2019,9,9);
  Date future_date{};
  future_date = 14 + d1;
  return 0;
}
```

Operatoröverlagring

En eller två parametrar?

- Vilken av sätten ska vi använda när?
- Det är okej att använda vilken ni vill. Det finns däremot stunder vi blir tvingade att använda alternativ #2 (att ta in två parametrar)

main.cc

```
int main()
{
  Date d1{2019,9,9};
  Date future_date{};
  future_date = 14 + d1;
  return 0;
}
```

Finns ingen inbyggd operator för int att addera ett Date-objekt och vi har inte tillgång till någon klass för int att skriva operatorn i

Operatoröverlagring

Addera int + egen datatyp

date.cc

```
int operator+(int lhs, Date const & rhs)
{
  return [addition av lhs och dagar i rhs];
}
```

main.cc

```
int main()
{
  Date d1{2019,9,9)
  Date future_date{};
  future_date = 14 + d1;
  return 0;
}
```

Operatoröverlagring

En eller två parametrar?

- Vilken av alternativen ska vi använda när?
- cout har datatypen ostream&
- returvärdet från (cout << d1) ska vara ostream&

main.cc

```
int main()
{
  Date d1{2019,9,9};
  cout << d1 << "\n";
  return 0;
}
```

Övning: Hur ser alternativ #1 och alternativ #2 ut för de båda operator << ? Vilket alternativ är genomförbart?

Agenda

1. Abstraktion
2. Klasser
3. Filuppdelning
4. Inkludering och kompilering (igen)
5. Operatoröverlagring
6. Funktionsöverlagring
7. Signalera fel
8. Mer om strömmar

LINKÖPINGS UNIVERSITET

Funktionsöverlagring

Vad händer när funktioner har samma namn?

- Funktioner kan ha samma namn i C++
- Parametrarna avgör vilken funktion som anropas
- Kompilatorn kommer att välja den funktion som ”matchar bäst” av funktioner med samma namn
- Returvärdet ingår inte i matchningen

Funktionsöverlagring

Vad händer när funktioner har samma namn?

```
int triangle_area(int base, int height);                    //version a
int triangle_area(int side1, int side2, int side3);         //version b
int triangle_area(int side1, int side2, double angle);      //verison c
int triangle_area(int side, double angle1, double angle2); //version d

triangle_area(1, 1) ;                                       //a anropas
triangle_area(1, 1, 1);                                     //b anropas
triangle_area(1, 1.0, 1.0);                                 //d anropas
triangle_area(1, 1, 1.0);                                   //c anropas
```

Funktioner - Bra att kunna inför Lab 1

Default argument

- Ibland vill vi att en funktion alltid ska få ett värde som argument, om inte den som anropar säger något annat.
- Vi kan därför sätta ett *default-argument* för våra parametrar.
- *OBS! Parametrar med default-argument måste ligga sist i parameterlistan.*
- Defaultvärdet specificeras endast vid *deklaration* av funktionen. Vid definitionen utelämnas det.

Funktioner - Bra att kunna inför Lab 1

Default argument exempel

```
void print_stars(int const n = 10);

int main()
{
  print_stars(20); // 20 stjärnor skrivs ut
  print_stars();   // 10 stjärnor skrivs ut (default)
  return 0;
}

void print_stars(int const n)
{
  for (int i{0}; i < n; ++i)
  {
    cout << '*';
  }
}
```

Undersök själv: antag att default-argument inte finns i språket. Kan du skapa motsvarande beteende med funktionsöverlagring (utan att duplicera någon kod)?

Agenda

1. Abstraktion
2. Klasser
3. Filuppdelning
4. Inkludering och kompilering (igen)
5. Operatoröverlagring
6. Funktionsöverlagring
7. Signalera fel
8. Mer om strömmar

LINKÖPINGS UNIVERSITET

Generera undantag (”kasta fel”)

Hur ska program signalera att det blivit fel?

- Ibland blir det fel
- Hur ska en programkomponent signalera till en annan att de värden komponenten fått att arbeta med inte är användbara?
  - **#1 Se till att endast användbar data går att skicka in genom smart val av datatyp**
  - **#2 Generera undantag (throw)**
  - #3 Kommunicera felkod via returvärde eller parameter (men lätt att anroparen helt enkelt glömmer kontrollera om fel uppstått!)
  - #4 Skriv ett felmeddelande i logfil och avsluta programmet

Generera undantag

Throw

```
#include <stdexcept>

class Card
{
public:
  Card(std::string suit, int value);

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

Card::Card(std::string suit, int value)
 : suit{suit}, value{value}
{
  if ( value < 0 || value > 13 )
  {
    throw std::logic_error{"Ogiltigt värde på kort"};
  }
}
```

```
#include <limits>

int main()
{
  Card power{
    "one to rule them all",
    numeric_limits<int>::max()
  };
}
```

Programmet kraschar om felet inte fångas upp:

```
terminate called after throwing an
instance of 'std::logic_error'
  what():  Ogiltigt värde på kort
Aborted (core dumped)
```

Fånga undantag

catch

Använda testramverket i Catch.hpp för att kontrollera att undantag genereras

```
TEST_CASE("Beskrivning ...")
{
  CHECK_THROWS( /* Anrop som genererar undantag */ );
}
```

Använda try-catch-satsen i C++ för att fånga upp och hantera undantag

```
int main()
{
  try // Var programmet ska leta efter undantag
  {
     // Satser som genererar undantag
  }
  catch(std::exception& e) // Vilket undantag programmet ska leta efter
  {
     // Hur undantaget ska åtgärdas
  }
}
```

Fånga undantag

catch

Använda testramverket i Catch.hpp för att kontrollera att undantag genereras

```
TEST_CASE("Card constructor")
{
  CHECK_THROWS(Card{ "one to rule them all", numeric_limits<int>::max()});
}
```

Använda try-catch-satsen i C++ för att fånga upp och hantera undantag

```
int main()
{
  try // Var programmet ska leta efter undantag
  {
     // Programmets normala function
     Card power{ "one to rule them all", numeric_limits<int>::max()};
  }
  catch(std::logic_error& e) // Vilket undantag programmet ska leta efter
  {
     // Hur undantaget ska åtgärdas
     std::cerr << e.what() << std::endl;
  }
}
```

Agenda

1. Abstraktion
2. Klasser
3. Filuppdelning
4. Inkludering och kompilering (igen)
5. Operatoröverlagring
6. Funktionsöverlagring
7. Signalera fel
8. Mer om strömmar

LINKÖPINGS UNIVERSITET

Datatyp för strömmar

Samlingstyp

- Alla typer av inströmmar är kompatibla med datatypen istream&
- Alla typer av utsrömmar är kompatibla med datatypen ostream&
- Dessa typer är lämpliga som paramerar till funktioner som ska hantera inmatning eller utmatning till valfri ström

Datatyp för strömmar

Samlingstyp

```
Person make_person(istream& is)
{
  string name{};
  int age;

  if ( iss >> name >> age )
  {
     cout << name << " är " << age << " år ung!" << endl;
  }
  return Person{name, age};
}

int main()
{
  string line{};
  while ( getline(ifs, line) )
  {
     istringstream iss{line};
     Person Q{ make_person( iss ) };
  }
  Person M{ make_person( cin ) };
}
```

Testa att programmet fungerar korrekt

Jobbigt för den som ska testa!

```
string read_date(istream&);

bool testcase_1()
{
   cout << "Mata in 1992-01-01 ";
   string date = read_date(cin);
   cout << date << endl;
   cout << "Kontrollera att 1992/01/01 skrevs ut och mata in 'y'";
   char assess;
   cin >> assess;
   return (assess == 'y');
}
```

Det måste finnas ett bättre sätt!

Testa att programmet fungerar korrekt

Ahhh! Bliss! Automatisk testning!

```
string read_date(istream&);

bool testcase_1()
{
   istringstream iss{"1992-01-01"};
   string date = read_date(iss);
   ostringstream oss;
   oss << date << endl;
   return (oss.str() == "1992/01/01");
}
```

"Fejka" cin och cout med hjälp av strängströmmar!

Seminarie 1 imorgon!

Andra typer av strömmar

Strängströmmar

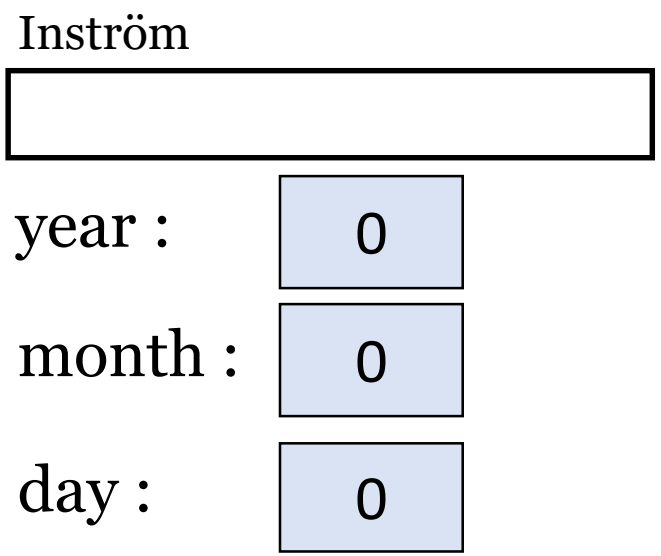
- Strängar är pilligt att arbeta med
  - Att få ut delar av en sträng
  - Konvertera från heltal till en sträng och formatera dem
- Tänk om vi kunde arbeta med strängar precis som vi arbetar med strömmar?

Andra typer av strömmar

Strängströmmar

- std::istringstream – en ”inström” vi kan lägga in en sträng i och läsa ifrån

```
int year{};
int month{};
int day{};
istringstream i_string_stream{”1992-02-27”};
i_string_stream >> year;
i_string_stream.get();
i_string_stream >> month;
i_string_stream.get();
i_string_stream >> day;
```

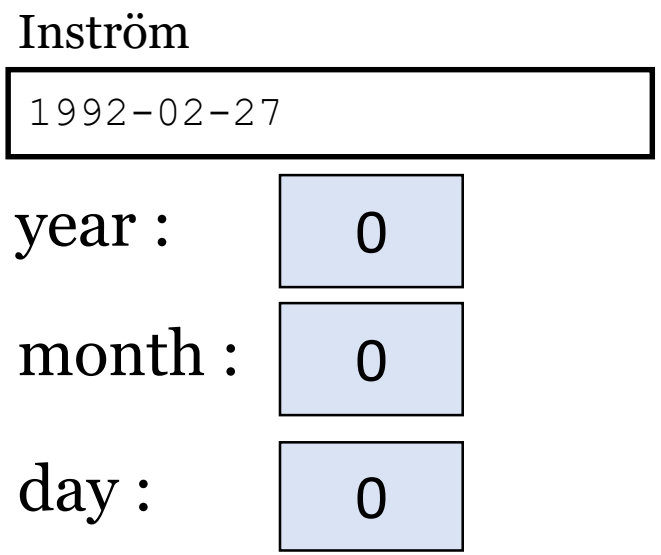


Andra typer av strömmar

Strängströmmar

- std::istringstream – en ”inström” vi kan lägga in en sträng i och läsa ifrån

```
int year{};
int month{};
int day{};
istringstream i_string_stream{”1992-02-27”};
i_string_stream >> year;
i_string_stream.get();
i_string_stream >> month;
i_string_stream.get();
i_string_stream >> day;
```

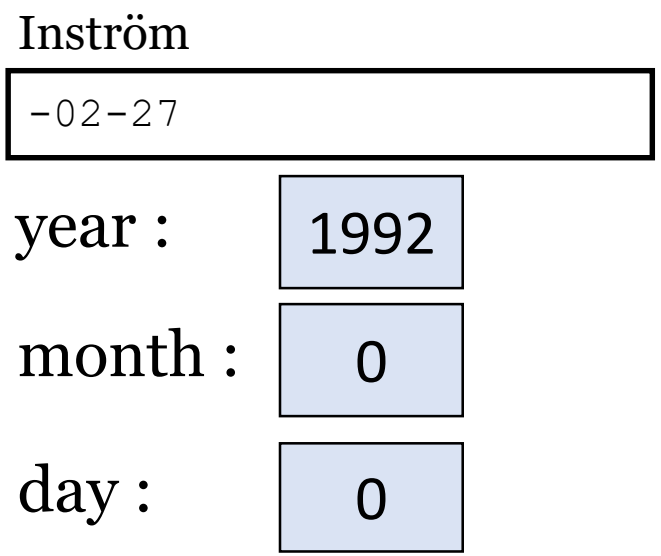


Andra typer av strömmar

Strängströmmar

- std::istringstream – en ”inström” vi kan lägga in en sträng i och läsa ifrån

```
int year{};
int month{};
int day{};
istringstream i_string_stream{”1992-02-27”};
i_string_stream >> year;
i_string_stream.get();
i_string_stream >> month;
i_string_stream.get();
i_string_stream >> day;
```

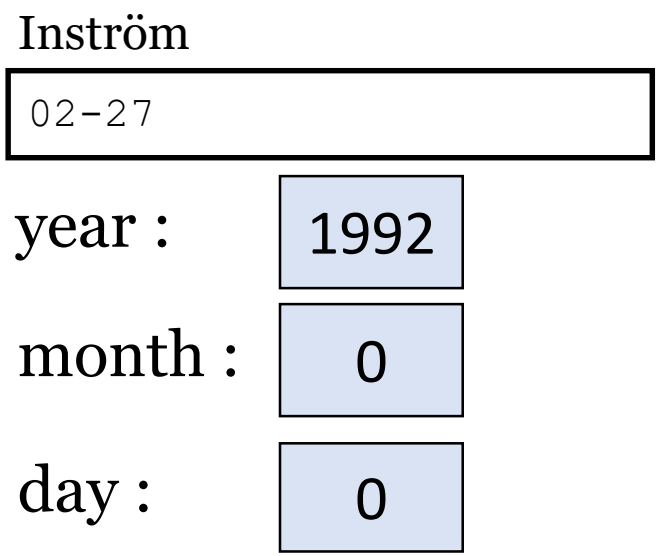


Andra typer av strömmar

Strängströmmar

- std::istringstream – en ”inström” vi kan lägga in en sträng i och läsa ifrån

```
int year{};
int month{};
int day{};
istringstream i_string_stream{”1992-02-27”};
i_string_stream >> year;
i_string_stream.get();
i_string_stream >> month;
i_string_stream.get();
i_string_stream >> day;
```

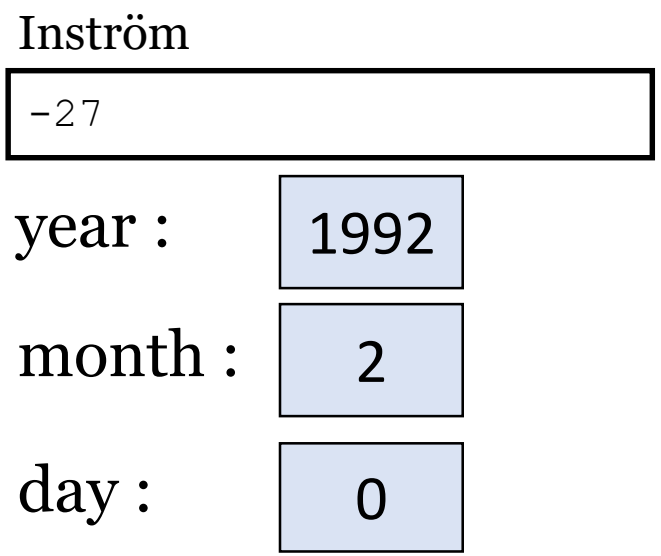


Andra typer av strömmar

Strängströmmar

- std::istringstream – en ”inström” vi kan lägga in en sträng i och läsa ifrån

```
int year{};
int month{};
int day{};
istringstream i_string_stream{”1992-02-27”};
i_string_stream >> year;
i_string_stream.get();
i_string_stream >> month;
i_string_stream.get();
i_string_stream >> day;
```

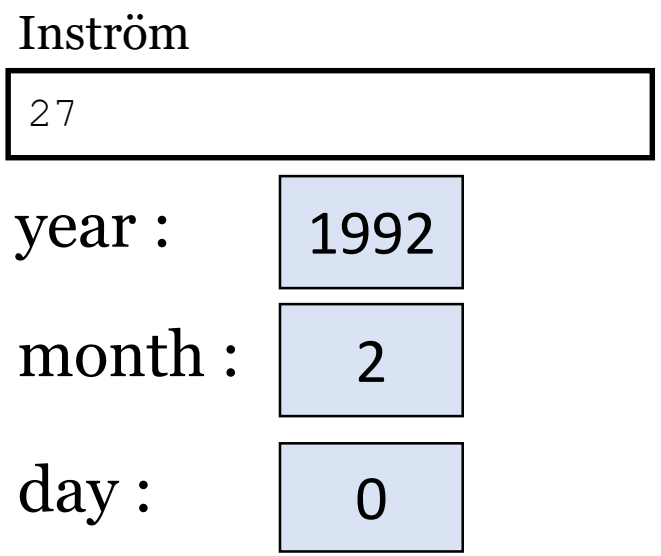


Andra typer av strömmar

Strängströmmar

- std::istringstream – en ”inström” vi kan lägga in en sträng i och läsa ifrån

```
int year{};
int month{};
int day{};
istringstream i_string_stream{”1992-02-27”};
i_string_stream >> year;
i_string_stream.get();
i_string_stream >> month;
i_string_stream.get();
i_string_stream >> day;
```

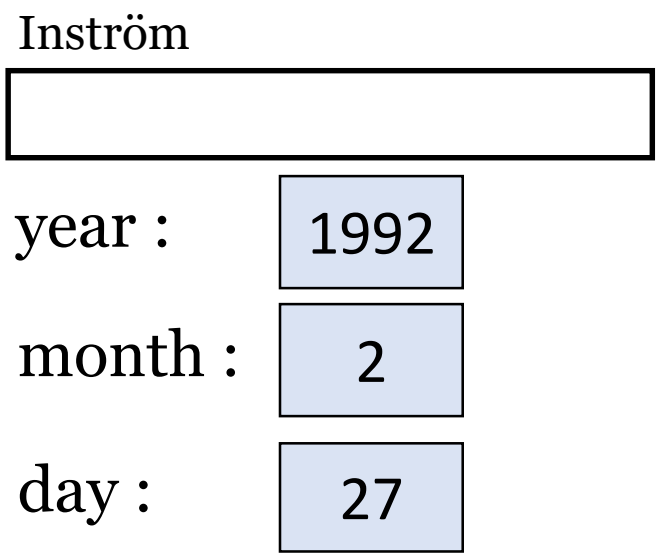


Andra typer av strömmar

Strängströmmar

- std::istringstream – en ”inström” vi kan lägga in en sträng i och läsa ifrån

```
int year{};
int month{};
int day{};
istringstream i_string_stream{”1992-02-27”};
i_string_stream >> year;
i_string_stream.get();
i_string_stream >> month;
i_string_stream.get();
i_string_stream >> day;
```

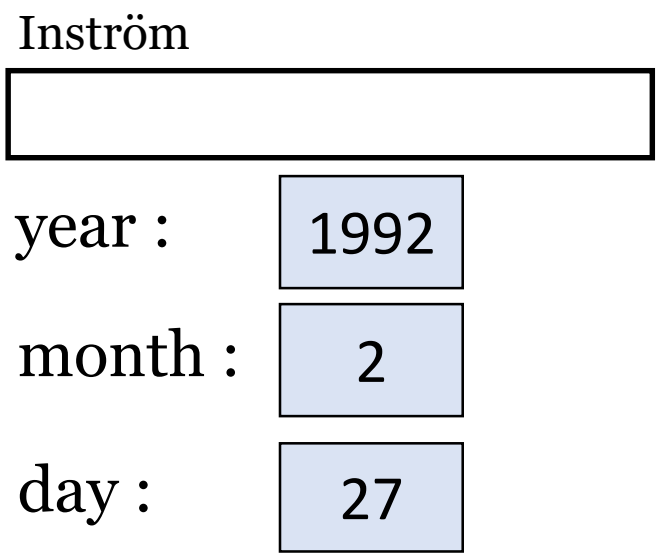


Andra typer av strömmar

Strängströmmar

- std::istringstream – en ”inström” vi kan lägga in en sträng i och läsa ifrån

```
int year{};
int month{};
int day{};
istringstream i_string_stream{”1992-02-27”};
i_string_stream >> year;
i_string_stream.get();
i_string_stream >> month;
i_string_stream.get();
i_string_stream >> day;
```



Andra typer av strömmar

Strängströmmar

- std::ostringstream – en ”utström” vi kan skriva ut till och hämta en sträng ifrån

```
int year{1992};
int month{2};
int day{27};
ostringstream o_string_stream{};
o_string_stream << setw(4) << setfill('0') << year;
o_string_stream << '-';
o_string_stream << setw(2) << setfill('0') << month;
o_string_stream << '-';
o_string_stream << setw(2) << setfill('0') << day;
```

Utström

year : 1992

month : 2

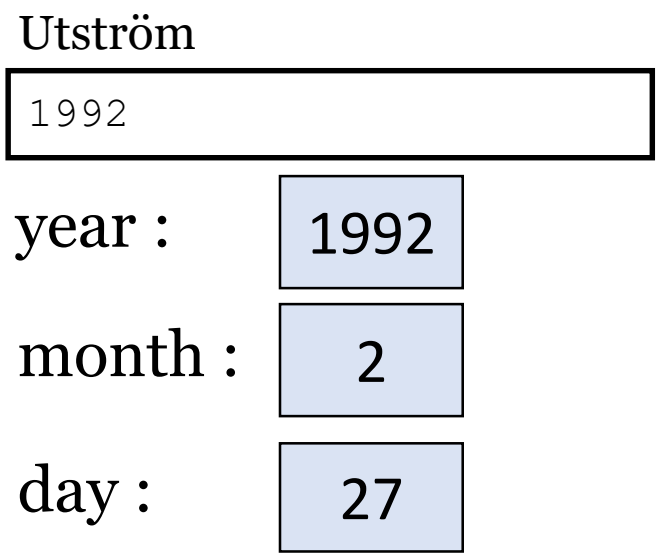
day : 27

Andra typer av strömmar

Strängströmmar

- std::ostringstream – en ”utström” vi kan skriva ut till och hämta en sträng ifrån

```
int year{1992};
int month{2};
int day{27};
ostringstream o_string_stream{};
o_string_stream << setw(4) << setfill('0') << year;
o_string_stream << '-';
o_string_stream << setw(2) << setfill('0') << month;
o_string_stream << '-';
o_string_stream << setw(2) << setfill('0') << day;
```



Andra typer av strömmar

Strängströmmar

- std::ostringstream – en ”utström” vi kan skriva ut till och hämta en sträng ifrån

```
int year{1992};
int month{2};
int day{27};
ostringstream o_string_stream{};
o_string_stream << setw(4) << setfill('0') << year;
o_string_stream << '-';
o_string_stream << setw(2) << setfill('0') << month;
o_string_stream << '-';
o_string_stream << setw(2) << setfill('0') << day;
```

Utström

```
1992-
```

year : 1992

month : 2

day : 27

Andra typer av strömmar

Strängströmmar

- std::ostringstream – en ”utström” vi kan skriva ut till och hämta en sträng ifrån

```
int year{1992};
int month{2};
int day{27};
ostringstream o_string_stream{};
o_string_stream << setw(4) << setfill('0') << year;
o_string_stream << '-';
o_string_stream << setw(2) << setfill('0') << month;
o_string_stream << '-';
o_string_stream << setw(2) << setfill('0') << day;
```

Utström

1992-02

year : 1992

month : 2

day : 27

Andra typer av strömmar

Strängströmmar

- std::ostringstream – en ”utström” vi kan skriva ut till och hämta en sträng ifrån

```
int year{1992};
int month{2};
int day{27};
ostringstream o_string_stream{};
o_string_stream << setw(4) << setfill('0') << year;
o_string_stream << '-';
o_string_stream << setw(2) << setfill('0') << month;
o_string_stream << '-';
o_string_stream << setw(2) << setfill('0') << day;
```

Utström

```
1992-02-
```

year : 1992

month : 2

day : 27

Andra typer av strömmar

Strängströmmar

- std::ostringstream – en ”utström” vi kan skriva ut till och hämta en sträng ifrån

```
int year{1992};
int month{2};
int day{27};
ostringstream o_string_stream{};
o_string_stream << setw(4) << setfill('0') << year;
o_string_stream << '-';
o_string_stream << setw(2) << setfill('0') << month;
o_string_stream << '-';
o_string_stream << setw(2) << setfill('0') << day;
```

Utström

1992-02-27

year : 1992

month : 2

day : 27

Andra typer av strömmar

Strängströmmar

- std::ostringstream – en ”utström” vi kan skriva ut till och hämta en sträng ifrån

```
int year{1992};
int month{2};
int day{27};
ostringstream o_string_stream{};
o_string_stream << setw(4) << setfill('0') << year;
o_string_stream << '-';
o_string_stream << setw(2) << setfill('0') << month;
o_string_stream << '-';
o_string_stream << setw(2) << setfill('0') << day;
string date{o_string_stream.str()};
```

Utström

1992-02-27

year :	1992
month :	2
day :	27

www.liu.se

LINKÖPINGS UNIVERSITET