TDP004 - Objektorienterd programmering

Arv, polymorfi, argument och felhantering
Handout

János Dani & Pontus Haglund

Institutionen för datavetenskap

LINKÖPING UNIVERSITY

1. Arv (Specialisering)
2. Bindning, static/dynamic
3. Cast av pekare och referenser
4. UML
5. Övriga hänsynstaganden
6. Smartpekare, interfaces, nvi
7. Argument till main

1 Arv (Specialisering)
2 Bindning, static/dynamic
3 Cast av pekare och referenser
4 UML
5 Övriga hänsynstaganden
6 Smartpekare, interfaces, nvi
7 Argument till main

LINKÖPING UNIVERSITY

Person

- Vi låtsas att en person har ett namn och ett telefonnummer.
- Hur kan vi representera en person?
- Med en klass så klart!

Person

```
// Person.h
class Person
{
  public:
  Person(std::string n, std::string p);

  std::string get_phone () const { return phone; }
  void print_info(ostream& os) const;

  private:
  string name;
  string phone;
};
```

```
// Person.cc
Person::Person(string n, string p):
name{n}, phone{p} {}

void Person::print_info(ostream& os) const
{
  os << "Name: " << name << '\n';
  os << "Phone: " <<  phone << endl;
}
```

Employee

Vi vill nu skapa en anställd. En anställd är en person med lön och arbetstelefon. Hur skapar vi den klassen när vi redan har en person?

- Vi kan kopiera och klistra till helt ny klass Employee. Det innebär att allt framtida arbete med Person-klassen manuellt måste dupliceras till Employee.
- Vi kan låta en ny Employee-klass innehålla en Person som datamedlem. Det innebär att vi måste skapa nya funktioner i Employee för att komma åt allt i Person.

Arv från / Specialisering av Person

Vi kan använda arv för att berätta för kompilatorn att en ny klass Employee automatiskt är allt som en Person är. Person blir **basklass** och Employee blir **härledd** klass (alternativt **subklass** eller **specialisering**):

```
class Employee : public Person
{
  public:
  Employee(string n, string p, string w, int s);
  string get_phone () const;
  string get_home_phone () const;
  string get_work_phone () const { return work_phone; }
  int salary () const;

  private:
  string work_phone;
  int salary;
};
```

Anrop av basklasskonstruktor från subklass

```
Employee(string n, string p, string w, int s)
: Person{n, p}, work_phone{w}, salary{s}
```

Datamedlemmarna **name** och **phone** är en del av basklassen. Det är därför praktiskt att anropa basklassens konstruktor för att initiera de datamedlemmarna

Anrop av basklasssens medlemsfunktion från subklass

En härledd klass har direkt alla datamedlemmar och medlemsfunktioner som finns i basklassen. Det går även att explicit ange att basklassens medlem ska användas. Men inte ens en härledd klass har tillgång till basklassens privata medlemmar!

```
string Employee::get_phone () const
{
  return Person::get_phone () + ", " + work_phone;
}

string Employee::get_home_phone () const
{
  return phone; // Compilation error , private to Person!
}
```

Åtkomst av datamedlemmar i subklass

Vi kan i basklassen ge härledda klasser tillgång till medlemmar med nyckelordet *protected*!

```
class Person
{
  public:
  ...

  private: // endast för klassen själv
  string name;

  protected: // endast för klassen själv och härledda klasser
  string phone;
};
```

Public/Private/Protected arv

Vi skrev ovan **public** när vi skapade subklassen. Varför?

```
class Employee : public Person
```

Vad gör det och andra nyckelord?

- public: medlemmar i basklassen behåller deklarerat skydd
- protected: public medlemmar i basklassen blir protected i härledd klass
- private: alla medlemmar i basklassen blir private i härledd klass

Public/Private/Protected arv forts

- Notera att privata medlemmar i basklassen aldrig är åtkomliga i subklassen.
- Lämnar du tomt väljs privat arv som standard.
- Var som vanligt **alltid** explicit

1 Arv (Specialisering)
2 Bindning, static/dynamic
3 Cast av pekare och referenser
4 UML
5 Övriga hänsynstaganden
6 Smartpekare, interfaces, nvi
7 Argument till main

Statisk bindning (static binding)

I C++ används statisk bindning om inget annat anges. Det innebär att datatypen som anges i programkoden avgör vilken funktion som anropas.

```
Person kim{"Kim", "2146"};
Employee sam{"Sam", "2490", "1234", 40000};
Person& ref{sam};

cout << kim.get_phone () << endl; // 2146
cout << sam.get_phone () << endl; // 2490, 1234
cout << sam.get_work_phone () << endl; // 1234
cout << ref.get_phone () << endl; // 2490
cout << ref.get_work_phone () << endl; // Compilation error!
sam.print_info(cout ); // Sams uppgifter skrivs ut på cout
kim.print_info(cout ); // Kims uppgifter skrivs ut på cout
```

Välj / välj bort specifika basklassmetoder

Det går att välja och vraka vilka medlemsfunktioner från basklasserna som ska finnas i den härledda klassen

```
// Tillfällig vikarie
class Substitute : public Employee
{
  public:
  using Employee :: Employee; // Ger konstruktor motsvarande Employee

  using Person :: get_phone (); // Väljer versionen i Person

  string get_work_phone () const = delete;
};
```

```
Substitute s{"Toni", "4711", "", 60000};

s.get_phone (); // Versionen i Person

s.get_work_phone (); // Kompileringsfel!
```

Subklass har alla datatyper

- En härledd klass är både sin egen datatyp och basklassens datatyp!
- En vikarie kan alltså användas som Person, Employee och Substitute.
- Vi bör dock tänka oss för vid tilldelningar!

```
Person p{"P", "1"};
Substitute s{"S", "2", "3", 0};

p = s; // Ok , men ger slicing
Employee& e = s; // Ok
Person* pp = &s; // Ok
Substitute v = e; // Compilation error!
```

Regler för typer

- En referens av typ T kan hänvisa till en typ härledd från T.
- En referens av typ T kan inte hänvisa till en bastyp av T.
- Ovanstående gäller även pekare!
- Ett objekt av typ T rymmer aldrig en härledd typ och kan inte lagra enbart en bastyp.

Exempel std::ostream

- std::ostream är basklassen för alla utströmmar
- Genom använda en std::ostream som parameter kan funktioner ta valfri härledd utströmtyp

```
void print_hello(ostream& os)
{
  os << "Hej" << endl;
}

ofstream ofs{"hello.txt"};
ostringstream oss;
print_hello(ofs);   // skriver till file
print_hello(oss);   // skriver till stringstream
print_hello(cout ); // skriver till standard output
print_hello(cerr ); // skriver till standard error
```

Exempel std::exception

- std::exception är basklassen för alla standardundantag
- Genom att fånga en basklassreferens kommer vi att fånga även alla undantag härledda från den fångade basklasstypen

```
try
{
  // Kod som kastar undantag ...
}
catch( logic_error& e)
{
  // Fångar logic_error i första hand
}
catch( exception& e )
{
  // Fångar klasser härledda från exception i andra hand
}
```

Flera olika typer av anställda i en lista?

- En pekare till Employee kan peka på härledda klasser
- Vilken funktion anropas?

```
vector <Employee*> v;
v.push_back(new Employee {...});
v.push_back(new Programmer {...});
v.push_back(new CEO {...});
v.push_back(new Consult {...});

for ( Employee const* e : v )
{
  cout << e->salary () << endl;
}
```

Implementation av programmer

```
class Programmer : public Employee
{
  public:
  Programmer (..., double e = 1.0)
  : Employee (...) , expertise{e} {}

  int salary () const
  {
    return salary * expertise;
  }

  private:
  double expertise;
};
```

Implementation av Consultant

```
class Consultant : public Employee
{
  public:
  Consultant (..., int p, int h)
  : Employee (...) , hour_pay{p}, hours_worked{h} {}

  int salary () const
  {
    return salary + hour_pay * hours_worked;
  }

  private:
  int hour_pay;
  int hours_worked;
};
```

Implementation av CEO

```
class CEO : public Employee
{
  public:
  CEO (..., int bonus = 1000000)
  Employee (...) , bonus{bonus} {}

  int bonus () const { return bonus; }

  int salary () const
  {
    return salary + bonus;
  }

  private:
  int bonus;
};
```

Löneberäkning av olika kategorier

- Statiskt bindning är default
- Med andra ord behöver vi göra något för att salary skall tas av subklassen!

```
vector <Employee*> v;
v.push_back(new Employee {...});
v.push_back(new Programmer {...});
v.push_back(new CEO {...});
v.push_back(new Consult {...});

for ( Employee const* e : v )
{
  // Employee::salary() anropas!
  cout << e->salary () << endl;
}
```

Dynamisk bindning till räddning!

virtual, override och final

```
class Employee : public Person
{
  // ...
  virtual int salary () const;
  // ...
  protected:
  int salary;
};

class CEO : public Employee
{
  // ...
  int salary () const override;
  // ...
};
```

Vtable

- Basklassen måste med virtual ange om härledda klasser kan ha en egen variant en funktion!
- Den härledda klassen ska med override ange att den ersätter basklassens variant, eller final.

Polymorfi

- När funktionen är *virtual* får vi dynamisk bindning (polymofi)
- Nu är det medlemsfunktionen som finns i det objekt som pekas ut som anropas

```
vector <Employee*> v;
v.push_back(new Employee {...});
v.push_back(new Programmer {...});
v.push_back(new CEO {...});
v.push_back(new Consult {...});

for ( Employee const* e : v )
{
  cout << e->salary () << endl;
}
```

Pekare och referenser - dynamisk typ

Vi kan endast ha dynamisk bindning(polymorfi) när vi anropar en virtual funktion från:

- en basklasspekare
- en basklassreferens

1 Arv (Specialisering)
2 Bindning, static/dynamic
3 Cast av pekare och referenser
4 UML
5 Övriga hänsynstaganden
6 Smartpekare, interfaces, nvi
7 Argument till main

dynamic_cast() ?

Ibland vill vi kunna ta reda på om en pekare egentligen pekar på en viss subklass:

```
for ( Employee const* e : v )
{
  CEO* ceo{ dynamic_cast <CEO*>(e) }; //Blir nullptr om typen är fel
  if ( ceo != nullptr )
  {
    cout << ceo->bonus ();
  }
  cout << e->salary () << endl;
}
```

Kommer ni ihåg dynamic_cast() ?

Pekare Ibland vill vi kunna ta reda på om en referens egentligen hänvisar till en viss subklass:

```
void print_salary( Employee const& e )
{
  cout << e.salary ();
  CEO& ceo{ dynamic_cast <CEO&>( e ) }; // may throw std::bad_cast
  cout << ceo.bonus ();
}

try
{
  CEO ceo{ ... };
  print_salary(ceo);
}
catch(std:: bad_cast const& e)
{
  cerr << e.what () << endl;
}
```

1 Arv (Specialisering)
2 Bindning, static/dynamic
3 Cast av pekare och referenser
4 UML
5 Övriga hänsynstaganden
6 Smartpekare, interfaces, nvi
7 Argument till main

LINKÖPING UNIVERSITY

Universal modelling language

- Grafiskt språk
- Används för att beskriva system
- bl.a. objektorienterade system

Vi ta upp 4 relationer:

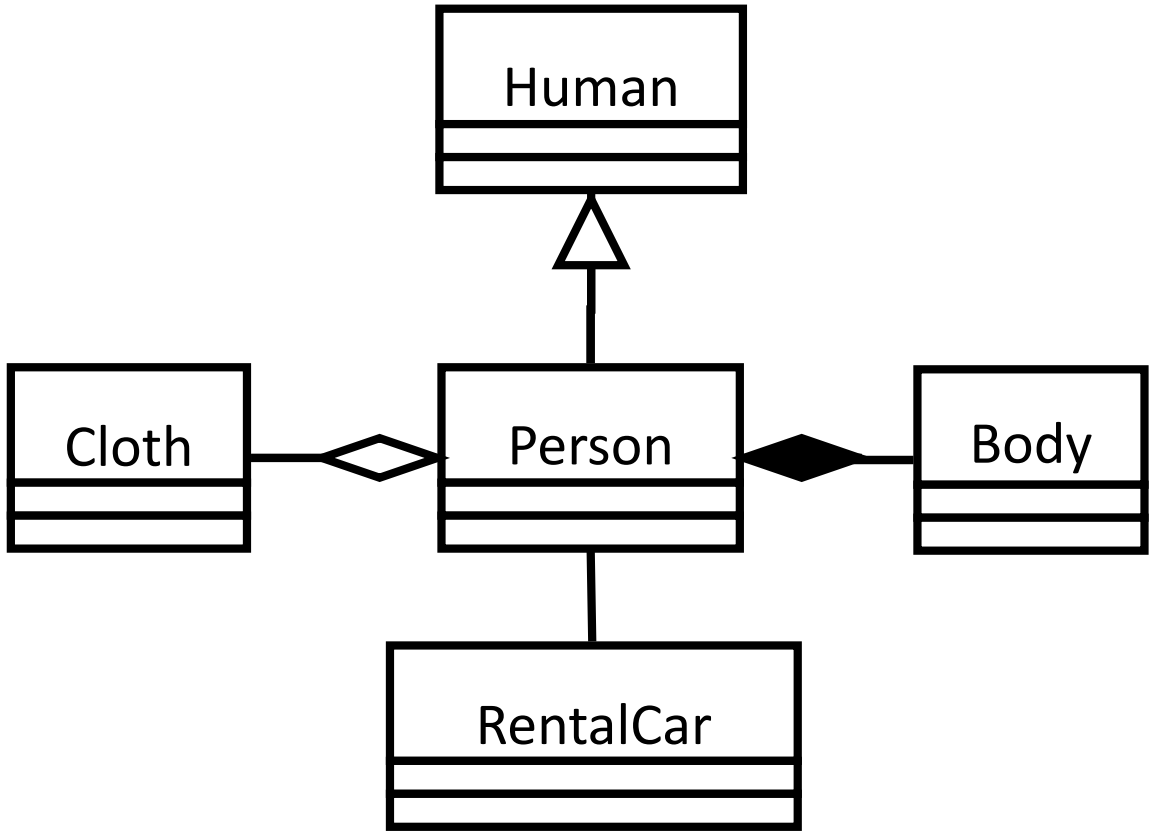
- Arv: En Person **är** en Människa med allt det innebär.
- Komposition: En Person **har** en Kropp.
- Aggregation: En Person **kan ha** Klädesplagg.
- Association: En Person **kan använda** en Hyrbil.

CRC-kort (analysmetod)

Vid objektorienterad analys kan vi använda CRC-kort för att komma fram till de klasser som behövs. Samtidigt definerar vi klassens ansvar och dess samarbetspartner. Man kan exempelvis använda postit-lappar och skriva:

- Class (klassens namn)
- Responsibility (Klassens ansvarsområde)
- Collaborators (samarbetspartners till klassen, hjälpklasser)

Uml resultat av analys

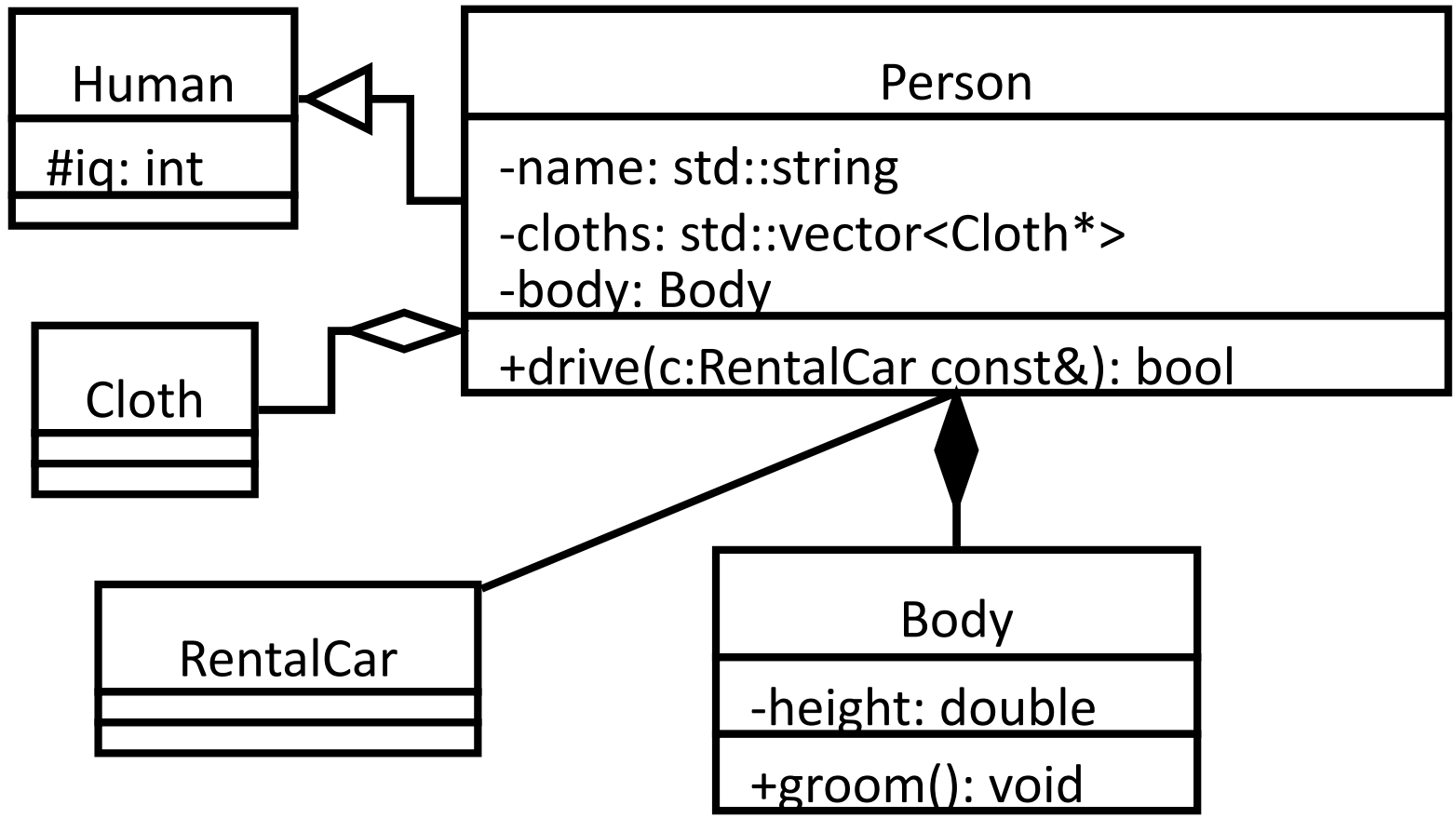


Mer detaljer

När de klasser vi behöver är indentifierade kan vi specificera dem i större detalj. Datamedlemmar, medlemsfunktioner ochd dess åtkomst specificeras:

- + public
- - private
- # protected
- : typ, returtyp eller datatyp

Mer detaljerat diagram



uml som kod

```
class Person : public Human // inheritance, is
{
  public:
  Person () : Human{}, body{}, clothing {} {}
  void put_on(Cloth* c);
  void drive(RentalCar &); // association, uses a

  private:
  Body body; // composition, always part of
  vector <Cloth*> clothing; // aggregation, can come and go
};
```

1 Arv (Specialisering)
2 Bindning, static/dynamic
3 Cast av pekare och referenser
4 UML
5 Övriga hänsynstaganden
6 Smartpekare, interfaces, nvi
7 Argument till main

LINKÖPING UNIVERSITY

Abstrakt klass **= 0**

En basklass är **abstrakt** om minst en medlemsfunktion är **pure virtual**. Om så är fallet måste alla subklasser implementera den funktionen.

```
class Employee : public Person // abstract class due to...
{
  // ...
  virtual int salary () const = 0; // pure virtual function
  // ...
};

Employee e{...}; // Compilation error , incomplete class
Employee* e{ new CEO {...}};
```

Minnehantering, destruktor

Som vanligt måste varje new motsvaras av en delete. Men hur anropar vi rätt destruktor?

```
vector <Employee*> v;
v.push_back(new Employee {...}); // Compilation error , incomplete class
v.push_back(new Programmer {...});
v.push_back(new CEO {...});
v.push_back(new Consult {...});

for ( Employee const* e : v )
{
  delete e; // Endast destruktorn för Employee!
}
```

Virtuell destruktor

Precis som medlemsfunktioner kan vara virtuella så vi hittar rätt funktion kan vi göra samma sak med destruktorn:

```
class Employee : public Person
{
  public:
  // ...
  virtual ~Employee () = default;
  // ...
}
```

```
for ( Employee const* e : v )
{
  delete e; // Varje destruktor anropas , den mest härledda först!
}
```

Egen undantagsklass

- Skapa egna undantag genom arv från undantag

```
class Stack_Error : public std:: logic_error
{
  public:
  // Använd logic_errors konstruktor
  using std::logic_error::logic_error;
};

class Code_Error : public std::exception
{
  public:
  // Egen konstruktor
  Code_Error(int code) noexcept : exception {}, code{code} {}

  const char* what () const noexcept { return ""; }
  int get_error_code () const { return code; }

  private:
  int code;
};
```

1 Arv (Specialisering)
2 Bindning, static/dynamic
3 Cast av pekare och referenser
4 UML
5 Övriga hänsynstaganden
6 Smartpekare, interfaces, nvi
7 Argument till main

LINKÖPING UNIVERSITY

Smarpekare

Finns två typer av smartpekare i c++17. unique_ptr och shared_ptr

```
{
  vector <unique_ptr <Employee >> v;
  v.push_back(unique_ptr <Employee >{new Employee {...}});//Samma som under
  v.push_back(make_unique <Employee >(...)); // men det här är säkrare
  v.push_back(make_unique <CEO >(...));
  // ...
} // Vectors destruktor kommer köra destruktorn för varje Employee
```

Interface

Ibland vill vi bara bestämma vilka operationer som ska gå att utföra på en viss typ av objekt. Vi bestämmer objektets gränssnitt (interface). I C++ kan vi då skapa en helt abstrakt klass (enbart virtual funktioner):

```
class Stack_Interface
{
  public:
  virtual void push(int value ) = 0;
  virtual void pop () = 0;
  virtual int& top () = 0;
  virtual int top() const = 0;
  virtual bool empty () const = 0;
};

class Linked_Stack : public Stack_Interface { ... }
class Vector_Stack : public Stack_Interface { ... }

// Denna funktion tar vilken stack som helst!
void use_stack(Stack_Interface& stack );
```

Varför ska dessa funktioner vara pure virtual?

Interface

Ibland vill vi bara bestämma vilka operationer som ska gå att utföra på en viss typ av objekt. Vi bestämmer objektets gränssnitt (interface). I C++ kan vi då skapa en helt abstrakt klass (enbart virtual funktioner):

```
class Stack_Interface
{
  public:
  virtual void push(int value ) = 0;
  virtual void pop () = 0;
  virtual int& top () = 0;
  virtual int top() const = 0;
  virtual bool empty () const = 0;
};

class Linked_Stack : public Stack_Interface { ... }
class Vector_Stack : public Stack_Interface { ... }

// Denna funktion tar vilken stack som helst!
void use_stack(Stack_Interface& stack );
```

För att de alltid ska implementeras av subklassen!

NVI icke-virtuellt gränssnitt

Ibland rekommenderas att virtual funktioner placeras privat. Det gör det enklare att ändra hur klassen löser problem internt utan att ändra på hur den används.

```
class Stack_Interface
{
  public:
  void push(int value );
  void pop ();
  //...

  private:
  // prefix undviker krock med publika medlemsfunktioner
  virtual void v_push(int value );
  virtual void v_pop ();
  //...
};
```

1 Arv (Specialisering)
2 Bindning, static/dynamic
3 Cast av pekare och referenser
4 UML
5 Övriga hänsynstaganden
6 Smartpekare, interfaces, nvi
7 Argument till main

Argument till main

Argument till main

```
int main(int argc, char* argv[])
{
  cout << "nr of arguments: " << argc << endl;
  cout << argv[0] << " : " << argv[1] << endl;
}
```

Argument till main

Typkonvertera argumenten

```
int main(int argc, char* argv[])
{
  cout << "nr of arguments: " << argc << endl;

  istringstream iss{argv[1]};
  double argument1{};
  iss >> argument1;
  cout << argument1 << endl;

  int argument2{ stoi(argv[2]) };
  cout << argument2 << endl;
}
```

Argument till main

Felhantering

```
int main(int argc, char* argv[])
{
  cout << "nr of arguments: " << argc << endl;
  try
  {
    int argument1{ stoi(argv[1]) };
  }
  catch (invalid_argument const& ia_error)
  {
    cerr << "Error: " << ia_error.what() << endl;
    return 1;
  }
  cout << argument1 << endl;
}
```

www.liu.se

LINKÖPING UNIVERSITY