TDDI82 - Objektorienterad problemlösning

Mallar

Christoffer Holm

Institutionen för datavetenskap

LINKÖPING UNIVERSITY

1 Namnrymder
2 Funktionsmallar
3 Klassmallar
4 Exempel

1 Namnrymder
2 Funktionsmallar
3 Klassmallar
4 Exempel

Namnrymder

Egen namnrymd

```
namespace NS
{
  class My_Class
  {
  };

  int my_fun(int x)
  {
    return x;
  }
}
```

Namnrymder

Egen namnrymd

```
namespace NS
{
  class My_Class;
  int my_fun(int x);
}
```

```
class NS::My_Class
{
};

int NS::my_fun(int x)
{
  return x;
}
```

Namnrymder

Egen namnrymd

```
// main.cc


int main()
{
  NS::My_Class m{};
  cout << NS::my_fun(3) << endl;
}
```

Namnrymder

Egen namnrymd

```
// main.cc
using namespace NS;

int main()
{
  My_Class m{};
  cout << my_fun(3) << endl;
}
```

1 Namnrymder
2 Funktionsmallar
3 Klassmallar
4 Exempel

Funktionsmallar

Exempel

```
int sum(vector<int> const& array)
{
  int result{};
  for (int const& e : array)
  {
    result += e;
  }
  return result;
}
```

Funktionsmallar

Exempel

```
double sum(vector<double> const& array)
{
  double result{};
  for (double const& e : array)
  {
    result += e;
  }
  return result;
}
```

Funktionsmallar

Exempel

```
string sum(vector<string> const& array)
{
  string result{};
  for (string const& e : array)
  {
    result += e;
  }
  return result;
}
```

Funktionsmallar

Mallar

```
template <typename T>
T sum(vector<T> const& array)
{
  T result{};
  for (T const& e : array)
  {
    result += e;
  }
  return e;
}
```

Funktionsmallar

Instansiering

```
int main()
{
  vector<int> v1{1, 2, 3};
  vector<double> v2{4.5, 6.7};

  cout << sum<int>(v1) << endl;
  cout << sum<double>(v2) << endl;
}
```

```
T sum(vector<T> const&)
int sum(vector<int> const&)
double sum(vector<double> const&)
```

Funktionsmallar

Instansiering

```
int main()
{
  vector<int> v1{1, 2, 3};
  vector<double> v2{4.5, 6.7};

  cout << sum(v1) << endl;
  cout << sum(v2) << endl;
}
```

```
T sum(vector<T> const&)
```

Funktionsmallar

Instansiering

```
int main()
{
  vector<int> v1{1, 2, 3};
  vector<double> v2{4.5, 6.7};

  cout << sum(v1) << endl;
  cout << sum(v2) << endl;
}
```

```
T sum(vector<T> const&)
```

Funktionsmallar

Instansiering

```
int main()
{
  vector<int> v1{1, 2, 3};
  vector<double> v2{4.5, 6.7};

  cout << sum(v1) << endl;
  cout << sum(v2) << endl;
}
```

```
T sum(vector<T> const&)
```

Funktionsmallar

Instansiering

```
int main()
{
  vector<int> v1{1, 2, 3};
  vector<double> v2{4.5, 6.7};

  cout << sum(v1) << endl;
  cout << sum(v2) << endl;
}
```

```
T sum(vector<T> const&)
```

Funktionsmallar

Instansiering

```
int main()
{
  vector<int> v1{1, 2, 3};
  vector<double> v2{4.5, 6.7};

  cout << sum(v1) << endl;
  cout << sum(v2) << endl;
}
```

```
T sum(vector<T> const&)
```

Funktionsmallar

Instansiering

```
int main()
{
  vector<int> v1{1, 2, 3};
  vector<double> v2{4.5, 6.7};

  cout << sum(v1) << endl;
  cout << sum(v2) << endl;
}
```

```
T sum(vector<T> const&)
int sum(vector<int> const&)
```

Funktionsmallar

Instansiering

```
int main()
{
  vector<int> v1{1, 2, 3};
  vector<double> v2{4.5, 6.7};

  cout << sum(v1) << endl;
  cout << sum(v2) << endl;
}
```

```
T sum(vector<T> const&)
int sum(vector<int> const&)
```

Funktionsmallar

Instansiering

```
int main()
{
  vector<int> v1{1, 2, 3};
  vector<double> v2{4.5, 6.7};

  cout << sum(v1) << endl;
  cout << sum(v2) << endl;
}
```

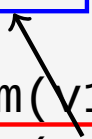
```
T sum(vector<T> const&)
int sum(vector<int> const&)
```

Funktionsmallar

Instansiering

```
int main()
{
  vector<int> v1{1, 2, 3};
  vector<double> v2{4.5, 6.7};

  cout << sum(v1) << endl;
  cout << sum(v2) << endl;
}
```



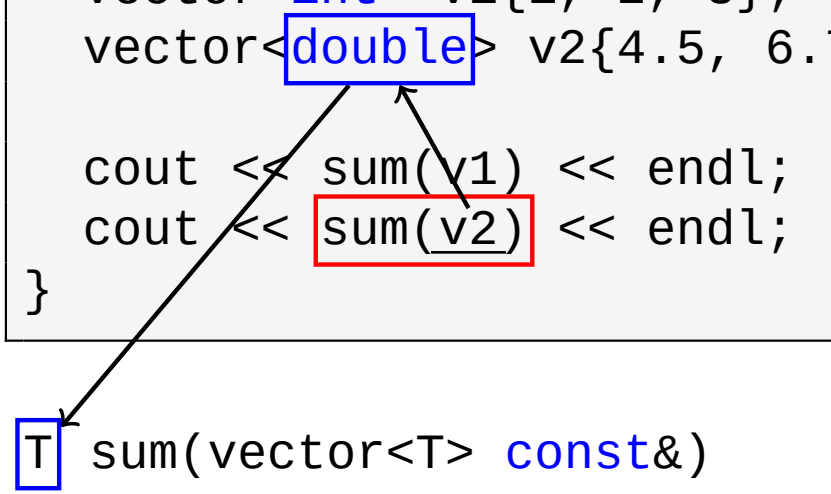
```
T sum(vector<T> const&)
int sum(vector<int> const&)
```

Funktionsmallar

Instansiering

```
int main()
{
  vector<int> v1{1, 2, 3};
  vector<double> v2{4.5, 6.7};

  cout << sum(v1) << endl;
  cout << sum(v2) << endl;
}
```



```
T sum(vector<T> const&)
int sum(vector<int> const&)
```

Funktionsmallar

Instansiering

```
int main()
{
  vector<int> v1{1, 2, 3};
  vector<double> v2{4.5, 6.7};

  cout << sum(v1) << endl;
  cout << sum(v2) << endl;
}
```

```
T sum(vector<T> const&)
int sum(vector<int> const&)
double sum(vector<double> const&)
```

Funktionsmallar

Fungerar ej

```
template <typename T>
T create()
{
  return T{};
}

int main()
{
  // Vad ska den skapa här?!
  cout << create() << endl;
  // här kommer vi skapa en double
  cout << create<double>() << endl;
}
```

Funktionsmallar

Standardtyp

```
template <typename T = int>
T create()
{
  return T{};
}

int main()
{
  // <...> saknas, så det blir standardtypen int
  cout << create() << endl;
  // här kommer vi skapa en double
  cout << create<double>() << endl;
}
```

Funktionsmallar

```
template <typename T, typename U>
T add(T a, U b)
{
  return a + b;
}
int main()
{
  // skriver ut 4
  cout << add<int, int>(1.2, 3.4) << endl;
  // skriver ut 3
  cout << add(1, 2.3) << endl;
  // skriver ut 3.3
  cout << add<double>(1, 2.3) << endl;
}
```

Funktionsmallar

Fler mallparametrar

```
template <typename Ret, typename T, typename U>
Ret add(T a, U b)
{
  return a + b;
}
int main()
{
  // funkar ej!
  cout << add(1, 2.3) << endl;
}
```

Funktionsmallar

Fler mallparametrar

```
template <typename Ret, typename T, typename U>
Ret add(T a, U b)
{
  return a + b;
}
int main()
{
  // ger svaret 3.3
  cout << add<double>(1, 2.3) << endl;
}
```

Funktionsmallar

Finns ett bättre sätt...

Funktionsmallar

`auto` som returtyp

```
template <typename T, typename U>
auto add(T a, U b)
{
  return a + b;
}
int main()
{
  // skriver ut 4
  cout << add<int, int>(1.2, 3.4) << endl;
  // skriver ut 3.3
  cout << add(1, 2.3) << endl;
}
```

Funktionsmallar

`auto` som returtyp

```
auto do_stuff(int x)
{
  if (x < 0)
  {
    return false; // bool
  }
  return x; // int
}
```

Funktionsmallar

`auto` som returtyp

```
auto do_stuff(int x)
{
  if (x < 0)
  {
    return false; // bool
  }
  return x; // int
}
```

Kompilerar ej

1 Namnrymder
2 Funktionsmallar
3 Klassmallar
4 Exempel

Klassmallar

Exempel

```
class Optional_Int
{
public:
  Optional_Int() = default;

  Optional_Int(int x);
  int& get();
  bool has_value() const;

private:
  int data{};
  bool active{};
};
```

```
Optional_Int::Optional_Int(int x)
  : data{x}, active{true}
{ }

// hämta värdet
int& Optional_Int::get()
{
  return data;
}

// kolla om det finns ett värde
bool Optional_Int::has_value() const
{
  return active;
}
```

Klassmallar

Exempel

```
class Optional_Double
{
public:
  Optional_Double() = default;

  Optional_Double(double x);
  double& get();
  bool has_value() const;

private:
  double data{};
  bool active{};
};
```

```
Optional_Double::Optional_Double(double x)
  : data{x}, active{true}
{ }

// hämta värdet
double& Optional_Double::get()
{
  return data;
}

// kolla om det finns ett värde
bool Optional_Double::has_value() const
{
  return active;
}
```

Klassmallar

Klassmallar

```
template <typename T>
class Optional
{
public:
  Optional() = default;
  Optional(T x)
    : data{x}, active{true}
  { }

  T& get() {
    return data;
  }

  bool has_value() const {
    return active;
  }
private:
  T data{};
  bool active{};
};
```

```
int main()
{
  // skapa en tom optional
  Optional<int> o1 {};

  // skapa en optional med 5
  Optional<int> o2 {5};

  // skapa en optional med 3.1
  Optional<double> o3 {3.1};

  if (o1.has_value())
  {
    cout << "Falskt!" << endl;
  }
  else if (o2.has_value())
  {
    cout << o2.get() << endl;
  }
}
```

Klassmallar

Instansiering

```
int main()
{
  // här måste vi ange typen
  Optional<int> o1 {};

  // funkar i C++17, ger T = int
  Optional o2 {5};

  // funkar alltid
  Optional<int> o3 {5};

}
```

Klassmallar

Uppdelning i h och cc-filer då?

```
template <typename T>
class Optional
{
public:
  Optional() = default;

  Optional(T x);
  T& get();
  bool has_value() const;

private:
  T data{};
  bool active{};
};
```

```
template <typename T>
Optional<T>::Optional(T x)
  : data{x}, active{true}
{ }

template <typename T>
T& Optional<T>::get()
{
  return data;
}

template <typename T>
bool Optional<T>::has_value() const
{
  return active;
}
```

Klassmallar

```
// optional.h
#ifndef OPTIONAL_H
#define OPTIONAL_H

template <typename T>
class Optional
{
public:
  // ...
  T& get();
  // ...
};

#include "optional.tcc"

#endif
```

```
// main.cc
#include "optional.h"
int main()
{
  // ...
}

// optional.tcc
// ...
template <typename T>
T& Optional<T>::get()
{
  return data;
}
// ...
```

Klassmallar

Uppdelning i h och cc-filer då?

```
// fil.h
#ifndef FIL_H
#define FIL_H

// deklaration
template <typename T,
          typename U>
auto sum(T a, U b);

#include "fil.tcc"
#endif FIL_H
```

```
// fil.tcc


//definition
template <typename T,
          typename U>
auto sum(T a, T b)
{
  return a + b;
}
```

1 Namnrymder
2 Funktionsmallar
3 Klassmallar
4 Exempel

Exempel

Ännu mer generell sum

```
int main()
{
  vector<int> v1{1, 2, 3};
  cout << sum(v1) << endl; // funkar

  vector<string> v2{"h", "e", "j"};
  cout << sum(v2) << endl; // funkar

  array<int, 3> a{1, 2, 3};
  cout << sum(a) << endl; // funkar ej
}
```

Exempel

Ännu mer generell `sum`

```
template <typename Container>
auto sum(Container const& c)
{
  /* värdetypen */ result{};
  for (auto const& e : c)
  {
    result += e;
  }
  return result;
}
```

Exempel

Ännu mer generell `sum`

```
template <typename Container>
auto sum(Container const& c)
{
  Container::value_type result{};
  for (auto const& e : c)
  {
    result += e;
  }
  return result;
}
```

Exempel

Ännu mer generell `sum`

```
template <typename Container>
auto sum(Container const& c)
{
  typename Container::value_type result{};
  for (auto const& e : c)
  {
    result += e;
  }
  return result;
}
```

Exempel

Ännu mer generell `sum`

- `typename Container::value_type` är även vår returtyp
- Därför kan det även vara bra att vara tydlig vad returtypen är genom att explicit ange det

Exempel

Ännu mer generell sum

```
template <typename Container>
typename Container::value_type //returtyp
sum(Container const& c)
{
  typename Container::value_type result{};
  for (auto const& e : c)
  {
    result += e;
  }
  return result;
}
```

Exempel

Iteratorer

```
template <typename Iterator>
auto sum(Iterator first, Iterator last)
{
  typename Iterator::value_type result{};
  for (auto it{first}; it != last; ++it)
  {
    result += *it;
  }
  return result;

}
```

Exempel

Iteratorer

```
int main()
{
  set<int> s{1, 2, 3};

  cout << sum(s) << endl;
  cout << sum(s.begin(), s.end()) << endl;

  vector<int> v{1, 2, 3};

  cout << sum(v) << endl;
  cout << sum(v.begin(), v.end()) << endl;
}
```

Exempel

Egen inretyp

```
template <typename T>
class My_Class
{
  using type = T;
  class My_Inner
  {
  };
};
```

```
template <typename T>
auto create_inner()
{
  return typename My_Class<T>::My_Inner{};
}

template <typename T>
typename My_Class<T>::type create_type()
{
  return typename My_Class<T>::type{};
}

int main()
{
  My_Class<int>::My_Inner my_inner{};
  My_Class<int>::type my_type{};
}
```

www.liu.se

LINKÖPING UNIVERSITY