

TDDI82 - Objektorienterad problemlösning

Mallar

Christoffer Holm

Institutionen för datavetenskap

- 1 Namnrymder
- 2 Funktionsmallar
- 3 Klassmallar
- 4 Exempel

- 1 Namnrymder
- 2 Funktionsmallar
- 3 Klassmallar
- 4 Exempel

Namnrymder

Egen namnrymd

- Namnrymder är bra om man har många saker med samma namn
- std är den mest kända namnrymden
- men det går också att skapa egna namnrymder där man kan placera sina saker för att hålla det separerade från resten av programmet

Namnrymder

Egen namnrymd

```
namespace NS
{
    class My_Class
    {
    };

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

Namnrymder

Egen namnrymd

- Man kan även här separerar deklaration och definition
- Det fungerar som vanligt men man lägger till namnrymden innan namnet på det man definierar

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

- Allt man kan göra med `std` kan man även göra med sina egna namnrymder
- Inklusive att inkludera hela namnrymden
- Även här gäller det att man inte göra `using namespace NS` i en h-fil

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

Exempel

- De alla är nästan samma kod
- jobbigt att behöva skriva det om och om igen
- Det vore bra om kompilatorn kunde lösa detta...

Funktionsmallar

Mallar

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

Funktionsmallar

Mallar

- Detta skapar en *funktionsmall*
- Det är **inte** en funktion
- en funktionsmall är en funktions generator...
- ... en mall som berättar för kompilatorn hur en funktion ska genereras!
- T är ett namn som berättar vart kompilatorn ska fylla ut med typer som användaren anger
- T kallas för en *mallparameter*

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

- Vi fyller i vad T är inom `<...>`
- Detta kallas *instansiering*
- Kompilatorn kommer instansiera (skapa) 2 separata funktioner:
- `int sum(vector<int> const&)` och
`double sum(vector<double> const&)`
- Vi kan även låta kompilatorn själv härleda vad T är...

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&)

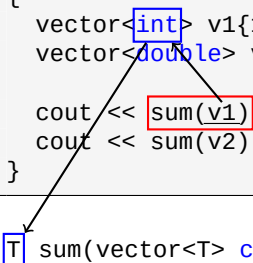
Funktionsmallar

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&)

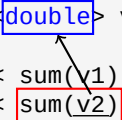
int sum(vector<int> const&)

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

Instansiering

- Kompilatorn är väldigt smart
- Den kan härleda vad T är även om det är "inbakat" i datatypen
- **MEN:** den kan endast härleda baserat på parametrar...

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

Flera mallparametrar

- Självklart kan vi ha fler än en mallparameter
- Detta ger oss två olika typer som kompilatorn kan fylla i
- Alla mallparametrar som förekommer som funktions parametrar kan kompilatorn härleda...

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

- Eftersom att returtypen är T kommer det alltid returnera samma typ som första parametern
- Detta gör så att om vi inte är försiktiga med vilka typer vi skickar som parametrar så kan det bli väldigt fel
- Ett sätt vi kan lösa detta på är att tvinga användaren ange vad denne vill att returtypen ska vara

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

Fler mallparametrar

- Det är viktigt att mallparametern som representerar returtypen är först
- kompilatorn kan härleda alla parametrar som är funktionsparameterar
- Men när användaren anger mallparametrar explicit innanför `< . . . >` så kommer kompilatorn sätta dessa från vänster till höger
- Eftersom att vi vill att användaren anger returtypen måste den därför vara först

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

- När vi anger `auto` som returtyp betyder det att kompilatorn ska härleda vad som returneras
- detta funkar endast om alla retursatser i funktionen alltid under alla omständigheter returnerar samma typ
- om vi har flera retursatser som returnerar olika kommer det inte att kompilera

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

optional

- Ibland kan det vara bra att ha en datatyp som ibland innehåller ett värde
- denna datatyp kallas en `optional`, den har antingen ett värde, som man kan plocka ut
- eller så finns inget värde och då får man ett odefinierat värde när man försöker plocka ut det
- Detta innebär att man alltid måste fråga huruvida värdet finns innan man plockar ut det

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

Generellt?

- Vi kan fortsätta skapa en optional för varje datatyp
- Detta är ju dock oerhört tidskrävande, speciellt om vi vill att det ska fungera för *alla* möjliga datatyper (inklusive typer skapade av andra)
- Nej, då är nog detta fel sätt att göra det på
- Vi använder *klassmallar* istället

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

Klassmallar

- *Klassmallar* fungerar som *funktionsmallar* med vissa skillnader
- En klassmall är en generator för typer
- Så `Optional` är inte en typ, medan exempelvis `Optional<int>` är det
- Fr.o.m. C++17 kan kompilatorn (för det mesta) härleda mallparametrarna från konstruktör anropen om varje mallparameter finns som parameter till konstruktorn

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?

- Men när vi skriver klasser ska vi separera deklaration och definition i olika filer?
- Eftersom att klassen är en mall, så beror varje medlemsfunktion på en mall parameter
- Därför måste vi för varje funktionskropp som vi definierar utanför klassdefinitionen ange att detta *beror* på en mall
- **Notera:** medlemsfunktioner i en klassmall är inte *nödvändigtvis* funktionsmallar

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

Uppdelning i h och cc-filer då?

- Det finns ett problem...
- När kompilatorn kompilerar en fil som använder `optional` måste den känna till *allt* om `optional` utan att kolla i filer som inte har inkluderats
- Detta innebär att den kommer inte se innehållet i `optional.cc`
- Därför måste hela definitionen av klassmallen finnas tillgänglig i h-filen
- Men det finns en lösning...

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?

- Vi kan inkludera implementationen i h-filen
- Det är rekommenderat att använda filändelsen `tcc` för implementation filen så att vi inte förvirrar den med `CC` filer
- Om vi försöker kompilera `tcc` filer kommer vi att få flera definitioner av samma medlemsfunktioner
- En definition från `main.cc` och en från `optional.tcc`
- Se därför till att inte kompilera `tcc` filer

Klassmallar

Detta går såklart även bra
med funktionsmallar

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

- Varje behållare har en inretyp som heter `value_type`
- detta är ett alias (ett annat namn) för den typ som behållaren innehåller
- för att komma åt värdetypen använder vi oss av detta

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

- Detta fungerar inte för att kompilatorn vet inte om `value_type` är en funktion, en variabel eller en datatyp. Den vet det först när vi har bestämt vad `T` är
- detta kallas att `value_type` är ett *dependent name*
- kompilatorn får inte acceptera detta, för att vad `value_type` är kan variera beroende på vad `T` är
- därför måste vi specificera att `value_type` är en datatyp med nyckelordet `typename`

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

Inretyper

- Det finns mycket i C++ som har inreklasser
- Standardbiblioteket kryllar av inretyper
- Exempelvis så innehåller iteratorer alltid också en `value_type`
- D.v.s. vilken datatyp som iteratorn pekar på
- Det finns även andra inretyper som behållare och iteratorer har
- Titta på cppreference.com för att se alla

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

- Självklart kan man skapa egna inretyper till sin klassmall
- Man kan antingen skapa alias m.h.a. `using`
- eller så kan man skapa inreklasser genom att deklarerera en klass inuti i sin klass
- Notera att inreklasser inte riktigt är klassmallar
- Men de är unika beroende på T fortfarande eftersom att de beror på en klassmall

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