TDDE18 & 726G77

Templates

Christoffer Holm

Department of Computer and information science

LINKÖPING
UNIVERSITY

1 Function templates
2 Class templates
3 Example
4 Namespaces

LINKÖPING
UNIVERSITY

Function templates

Example

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

Function templates

Example

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

Function templates

Example

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

Function templates

Example

- They all are nearly identical pieces of code
- It is very tedious to have to write the same code again and again
- Would be nice if the compiler could do this for us...

Function templates

Templates

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

Function templates

Templates

- This creates a *function template*
- It is **not** a function
- A function template is a function generator...
- ... a template that tells the compiler how a certain type of function can be generated!
- `T` is a name that tells the compiler where it should fill in the data type that the user specifies
- `T` is called a *template parameter*

Function templates

Instantiation

```
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&)
```

Function templates

Instantiation

- We specify what `T` is inside the `<...>`
- This is called *instantiation*
- The compiler will instantiate (create) 2 seperate functions:
- `int sum(vector<int> const&)` and `double sum(vector<double> const&)`
- We can also allow the compiler to deduce what `T` must be...

Function templates

Instantiation

```
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&)
```

Function templates

Instantiation

```
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&)`

Function templates

Instantiation

```
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&)
```

Function templates

Instantiation

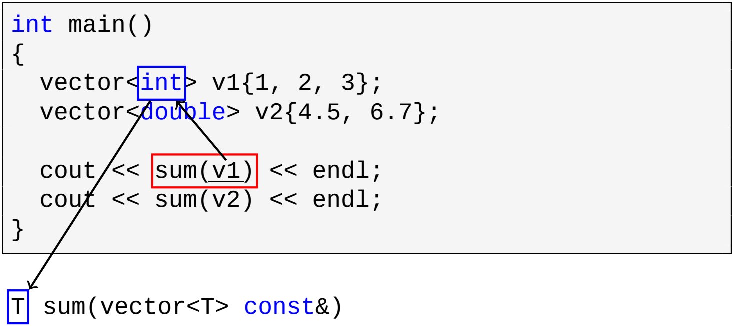
```
int main()
{
  vector<int> v1{1, 2, 3};
  vector<double> v2{4.5, 6.7};

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

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

Function templates

Instantiation



```
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&)
```

Function templates

Instantiation

```
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&)
```

Function templates

Instantiation

```
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&)
```

Function templates

Instantiation

```
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&)
```

Function templates

Instantiation

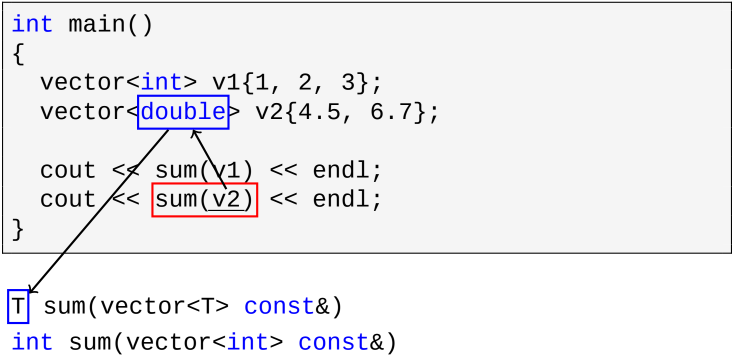
```
int main()
{
  vector<int> v1{1, 2, 3};
  vector<double> v2{4.5, 6.7};

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

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

Function templates

Instantiation



```
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&)
```

Function templates

Instantiation

```
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&)
```

Function templates

Instantiation

- The compiler is pretty smart
- It can deduce what `T` is even if it is embedded inside another data type
- **BUT**: it can only deduce based on parameter...

Function templates

Doesn’t work

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

int main()
{
  // what should it create?!
  cout << create() << endl;
  // create a double
  cout << create<double>() << endl;
}
```

Function templates

Standard type

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

int main()
{
  // <...> is missing, so it defaults to int
  cout << create() << endl;
  // create a double
  cout << create<double>() << endl;
}
```

Function templates

Multiple template parameters

- We can have multiple template parameters
- This gives us two (or more) different types the compiler can substitute
- All template parameters that occur as function parameters can the compiler deduce...

Function templates

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

Function templates

Multiple template parameters

- The return type is `T` so the function will always return the same data type as the first parameter
- If we are not careful in which order we pass the types we can get inaccurate results
- One way to solve this is by forcing the user to specify the return type

Function templates

Multiple template parameters

```
template <typename Ret, typename T, typename U>
Ret add(T a, U b)
{
  return a + b;
}
int main()
{
  // Doesn't work!
  cout << add(1, 2.3) << endl;
}
```

Function templates

Multiple template parameters

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

Function templates

Multiple template parameters

- It is important that the template parameter for the return type occurs first
- The compiler can deduce all parameters that are function parameters
- But when a template parameter is specified explicitly inside <. . .> then the compiler will substitute these from left to right
- Since we want the user to specify the return type but nothing else we must therefore place it at the start

Function templates

There is a better way...

Function templates

`auto` as return type

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

Function templates

`auto` as return type

- When we specify `auto` as the return type we tell the compiler to deduce the return type
- This will only work if all `return`-statements in the function always return the same type
- If we have multiple `return`-statements that return different types we will get a compile error

Function templates

`auto` as return type

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

Function templates

`auto` as return type

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

Compile error

Class templates

`optional`

- In some cases it is nice to have a data type that might contain a value (but doesn't have to)
- This data type, called `optional`, it either has a value that can be retrieved
- Or no value at all (giving us some kind of error when we try to retrieve it)
- This means that we always have to check first whether or not it has a value before we retrieve it

Class templates

Example

```
class Optional_Int
{
public:
  // Set data to nullptr
  Optional_Int() = default;

  Optional_Int(int x);

  int& get();

  bool has_value() const;

private:
  unique_ptr<int> data{};
};
```

```
Optional_Int::Optional_Int(int x)
  : data{make_unique<int>(x)}
{ }

// Retrieve the value
int& Optional_Int::get()
{
  return *data;
}

// Check if there is a value
bool Optional_Int::has_value() const
{
  return data != nullptr;
}
```

Class templates

Example

```cpp
class Optional_Double
{
public:
  // Set data to nullptr
  Optional_Double() = default;

  Optional_Double(double x);

  double& get();

  bool has_value() const;

private:
  unique_ptr<double> data{};
};
```

```cpp
Optional_Double::Optional_Double(double x)
  : data{make_unique<double>(x)}
{ }

// Retrieve the value
double& Optional_Double::get()
{
  return *data;
}

// Check if there is a value
bool Optional_Double::has_value() const
{
  return data != nullptr;
}
```

Class templates

General?

- We can continue creating an optional for each data type
- This is very time consuming, especially if we want it to work for *all* possible data types
- This is probably the wrong way to solve this problem
- Let’s use *class templates* instead

Class templates

```cpp
template <typename T>
class Optional
{
public:
  Optional() = default;
  Optional(T x)
    : data{make_unique<T>(x)}
  { }

  T& get()
  {
    return *data;
  }

  bool has_value() const
  {
    return data != nullptr;
  }
private:
  unique_ptr<T> data{};
};
```

```cpp
int main()
{
  // create an empty optional
  Optional<int> o1 {};

  // create an optional 5
  Optional<int> o2 {5};

  // create an optional 3.1
  Optional<double> o3 {3.1};

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

Class templates

Class templates

- *Class templates* work like *function templates* with some differences
- A class template is a generator for data types (classes)
- So `Optional` is not a type, while for example `Optional<int>` is
- From C++17 and onwards the compile can (for the most part) deduce template parameters from the constructor call if each template parameter is present as a parameter to the constructor

Class templates

Instantiation

```
int main()
{
  // We must specify the type
  Optional<int> o1 {};

  // works in C++17, gives us T = int
  Optional o2 {5};

  // always works
  Optional<int> o3 {5};

}
```

Class templates

what about h and cc-files?

- But when we write classes are we not supposed to separate declaration and definition into different files?
- Since the class is a template the member functions depend on template parameters
- Therefore we must specify this for each member function implementation
- **Note:** member functions in a class template is not necessarily a function template

Class templates

what about h and cc-files?

```
template <typename T>
class Optional
{
public:
  // Set data to nullptr
  Optional() = default;

  Optional(T x);

  T& get();

  bool has_value() const;

private:
  unique_ptr<T> data{};
};
```

```
template <typename T>
Optional<T>::Optional(T x)
  : data{make_unique<T>(x)}
{ }

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

template <typename T>
bool Optional<T>::has_value() const
{
  return data != nullptr;
}
```

Class templates

what about h and cc-files?

- There is a problem...
- When the compiler compiles a file that uses `Optional` it must know *everything* about `Optional` without checking files that haven't been included
- This means the compiler won't see the content of `optional.cc`
- Therefore the entire class template must be available in the h-file, implementation and all
- There is a solution...

Class templates

```
// 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;
}
// ...
```

Class templates

what about h and cc-files?

- We can include the implementation in the h-file
- It is recommended to use the file extension `tcc` for the implementation file so we don't confuse them with `cc` files
- If we try to compile `tcc` files then we will get multiple definitions of the same member function
- One from `main.cc` and one from `optional.tcc`
- Make sure to not compile `tcc` files

Class templates

Also works with function tenplates

Class templates

what about h and cc-files?

```
// file.h
#ifndef FILE_H
#define FILE_H
// deklaration
template <typename T,
          typename U>
auto sum(T a, U b);

#include "file.tcc"
#endif//FILE_H
```

```
// file.tcc


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

Example

Even more general `sum`

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

  vector<string> v2{"h", "i", "!"};
  cout << sum(v2) << endl; // works

  array<int, 3> a{1, 2, 3};
  cout << sum(a) << endl; // doesn't work
}
```

Example

Even more general `sum`

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

Example

Even more general `sum`

- Each container has an inner type called `value_type`
- This is an alias (different name) for the type the container holds
- Let’s use this alias

Example

Even more general `sum`

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

Example

Even more general `sum`

- This doesn't work because the compiler doesn't know whether or not `value_type` is a function, a variable or a data type. This is first known once `Container` has been substituted with a type
- We say that `value_type` is a *dependent name*
- The compiler cannot accept this, because what `value_type` is can vary depending on `Container`
- Therefore we must specify that `value_type` is a data type with the keyword `typename`

Example

Even more general `sum`

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

Example

Even more general `sum`

- `typename Container::value_type` is also our return type
- It can be good to be clear what the return type is by explicitly specify it

Example

Even more general `sum`

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

Example

Inner types

- There is a lot in C++ that has inner types
- The standard library is filled with inner types
- For example: iterators always contains a `value_type`
- That is which data type the iterator points to
- There are also other inner types that containers and iterators have, look at cppreference.com

Example

Iterators

```
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;

}
```

Example

Iterators

```
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;
}
```

Example

Custom inner type

- You can create your own inner types
- You can either create an alias with `using`
- Or you can create an inner type by creating an inner class
- Note that inner classes aren't class templates
- But they are unique for each `T` (i.e. each instantiation of your class template will have its own version of the inner class)

Example

Custom inner type

```
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{};
}
```

1 Function templates
2 Class templates
3 Example
4 Namespaces

Namespaces

Custom namespace

- Namespaces are good if you have many things with the same name in different contexts
- `std` is the most known namespace
- You can also create custom namespaces where you place your functions/classes to keep them separated from the rest of the code

Namespaces

Custom namespace

```
namespace NS
{
  class My_Class
  {
  };

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

Namespaces

Custom namespace

- You can separate declaration and definition
- It works as normal but you add the namespace before the name

Namespaces

Custom namespace

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

```
class NS::My_Class
{
};

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

Namespaces

Custom namespace

- Everything you can do with `std` you can do with your own namespace
- Including importing the entire namespace with `using namespace NS`
- (Don't import it in h-files)

Namespaces

Custom namespace

```
// main.cc


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

Namespaces

Custom namespace

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

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

www.liu.se

LINKÖPING
UNIVERSITY