TDDD38 - Advanced programming in C++

Templates II

Christoffer Holm

Department of Computer and information science

LINKÖPING UNIVERSITY

LINKÖPING UNIVERSITY

LINKÖPING UNIVERSITY

Class Templates

Basic Class Templates

```
#include <cstddef> // size_t

template <typename T, size_t N>
class Array
{
public:
  static size_t size()
  {
    return N;
  }

  T& operator[](size_t i)
  {
    return data[i];
  }
private:
  T data[N]{};
};
```

Class Templates

Basic Class Templates

- class templates are not classes;
- they are templates for generating classes during *instantiation*;
- member functions are not necessarily function templates; they are only generated whenever the class template is instantiated.

Class Templates

Member Functions

array.h

```
#include <cstddef> // size_t

template <typename T, size_t N>
class Array
{
public:
  static size_t size();
  T& operator[](size_t i);
private:
  T data[N]{};
};

#include "array.tcc"
```

Class Templates

Member Functions

array.h

```
#include <cstddef> // size_t

template <typename T, size_t N>
class Array
{
public:
  static size_t size();
  T& operator[](size_t i);
private:
  T data[N]{};
};

#include "array.tcc"
```

array.tcc

```
template <typename T, size_t N>
size_t Array<T, N>::size()
{
  return N;
}

template <typename T, size_t N>
T& Array<T, N>::operator[](size_t i)
{
  return data[i];
}
```

Class Templates

Member Functions

- It can be useful to separate the class template definition and the member function definitions;
- just as with function templates, the compiler must know everything about a class template before it is able instantiate the class;
- because of this we should include the member function definition file in the header file.

Class Templates

Member Functions

- Member functions depend on the class template;
- must use instantiation arguments in the qualified name;
- therefore we must use templates to specify these instantiation arguments.

Class Templates

Instantiation

```
#include "array.h"

int main()
{
  Array<int, 3> arr;
  for (size_t i{0}; i < arr.size(); ++i)
  {
    arr[i] = i;
  }
}
```

Class Templates

Member Function Templates

array.h

```
#include <cstddef> // size_t

template <typename T, size_t N>
class Array
{
public:
  // ...
  template <size_t M>
  Array<T, N+M> concat(Array<T, M> const& other);
  // ...
};

#include "array.tcc"
```

Class Templates

Member Function Templates

array.tcc

```
// ...
template <typename T, size_t N>
template <size_t M>
Array<T, N+M> Array<T, N>::concat(Array<T, M> const& other)
{
  Array<T, N+M> result;
  for (size_t i{0}; i < N; ++i)
  {
    result[i] = data[i];
  }
  for (size_t i{0}; i < M; ++i)
  {
    result[N + i] = other[i];
  }
  return result;
}
// ...
```

Class Templates

Instantiation

- Instantiating a class template will generate a distinct class for each set of unique template parameters;
- since the template parameters are bound to the type we can then proceed to use the member functions as normal, no need to supply the template parameters.

Class Templates

Specialization

```
template<>
class Array<int, 0>
{
public:
  static size_t size()
  {
    return 0;
  }
  int& operator[](size_t i)
  {
    throw std::out_of_range{"No elements"};
  }
};
```

Class Templates

Specialization

- Just like with template functions (you will be hearing this alot), you can specialize your class templates for specific template parameters;
- this is used a lot more than explicit function template specialization, since there is no way to overload classes;
- whenever a class template is instantiated the compiler will look for specializations;
- **Warning:** the specialization should have been declared before the instantiation.

Class Templates

Partial Specialization

```
template<typename T>
class Array<T, 0>
{
public:
  static size_t size()
  {
    return 0;
  }
  T& operator[](size_t i)
  {
    throw std::out_of_range{"No elements"};
  }
};
```

Class Templates

Partial Specialization

- One thing that class templates can do which function templates are unable to do, is *partial specialization*;
- this allows you to only specialize a subset of the template parameters;
- it can also be used to narrow the possibilites of a template parameter (this will come later).

Class Templates

Partial Specialization Restrictions

- Cannot be identical to the primary templates parameter list;
- Must be more *specialized* than the primary template;
- No default arguments are allowed;
- Each nontype argument must be *deducible* by the compiler;
- Nontype parameters cannot be specialized if other template parameters depend on them.

LINKÖPING UNIVERSITY

Variadic Templates

Initialization of `Array`

```
#include "array.h"

int main()
{
  Array<int, 3> arr{1,2,3};
}
```

Variadic Templates

Variadic Templates

array.h

```
#include <cstddef> // size_t

template <typename T, size_t N>
class Array
{
public:
    Array() = default;

    template <typename... Ts>
    Array(Ts... list)
        : data{list...}
    { }
    // ...
};

#include "array.tcc"
```

Variadic Templates

Parameter Pack

- `typename... Ts`
- `Ts... list`
- `list...`

Variadic Templates

Parameter Pack

- `typename... Ts`
  - *Template parameter pack*;
  - a template parameter which takes zero or more arguments.
- `Ts... list`
- `list...`

Variadic Templates

Parameter Pack

- `typename... Ts`
- `Ts... list`
    - *Function parameter pack*;
    - Function parameter that takes zero or more arguments.
- `list...`

Variadic Templates

Parameter Pack

- `typename... Ts`
- `Ts... list`
- `list...`
  - *Parameter pack expansion*;
  - Expand to a comma-separated list of zero or more values;
  - the type of these values will correspond to the types in `Ts...`
  - only works inside *lists*;

Variadic Templates

Parameter Pack

```
template <typename T, size_t N>
template <typename... Ts>
Array<T, N>::Array(Ts... list)
: data{list...}
{ }

int main()
{
  Array<int, 3> arr{1,2,3};
}
```

Variadic Templates

Parameter Pack

```
template <typename... Ts>
Array<int, 3>::Array(Ts... list)
: data{list...}
{ }

int main()
{
  Array<int, 3> arr{1,2,3};
}
```

Variadic Templates

Parameter Pack

```
template <typename T1, typename T2, typename T3>
Array<int, 3>::Array(Ts... list)
: data{list...}
{ }

int main()
{
  Array<int, 3> arr{1,2,3};
}
```

Variadic Templates

Parameter Pack

```
template <typename T1, typename T2, typename T3>
Array<int, 3>::Array(T1 l1, T2 l2, T3 l3)
: data{list...}
{ }

int main()
{
  Array<int, 3> arr{1,2,3};
}
```

Variadic Templates

Parameter Pack

```
template <typename T1, typename T2, typename T3>
Array<int, 3>::Array(T1 l1, T2 l2, T3 l3)
: data{l1, l2, l3}
{ }

int main()
{
  Array<int, 3> arr{1,2,3};
}
```

Variadic Templates

Parameter Pack

```
Array<int, 3>::Array(int l1, int l2, int l3)
: data{l1, l2, l3}
{ }

int main()
{
  Array<int, 3> arr{1,2,3};
}
```

Variadic Templates

Parameter Pack

```
Array<int, 3>::Array(int l1, char const* l2, int l3)
: data{l1, l2, l3}
{ }

int main()
{
  Array<int, 3> arr{1,"2",3};
}
```

Variadic Templates

Parameter Pack

```
Array<int, 3>::Array(int l1, char const* l2, int l3)
: data{l1, l2, l3}
{ }

int main()
{
  Array<int, 3> arr{1,"2",3};
}
```

Compile Error

Variadic Templates

Parameter Pack

```
Array<int, 3>::Array(int l1, int l2, int l3, int l4)
: data{l1, l2, l3, l4}
{ }

int main()
{
  Array<int, 3> arr{1,2,3,4};
}
```

Variadic Templates

Parameter Pack

```
Array<int, 3>::Array(int l1, int l2, int l3, int l4)
: data{l1, l2, l3, l4}
{ }

int main()
{
  Array<int, 3> arr{1,2,3,4};
}
```

Compile Error

LINKÖPING
UNIVERSITY

Template Usage & Error checking

Variadic Recursion

array.h

```
#include <cstddef> // size_t

template <typename T, size_t N>
class Array
{
public:
    // ...
    template <typename... Ts>
    void set(Ts... list);
    // ...
};

#include "array.tcc"
```

Template Usage & Error checking

Variadic Recursion

array.h

```
#include <cstddef> // size_t

template <typename T, size_t N>
class Array
{
public:
    // ...
    template <typename... Ts>
    void set(Ts... list);
    // ...
};

#include "array.tcc"
```

array.tcc

```
// ...
template <typename T, size_t N>
template <typename... Ts>
void Array<T, N>::set(Ts... list)
{
  // ???
}
// ...
```

Template Usage & Error checking

Variadic Recursion

- There is a way to unpack the parameter pack;

Template Usage & Error checking

Variadic Recursion

- There is a way to unpack the parameter pack;
- with recursion!

Template Usage & Error checking

Variadic Recursion

```
template <typename... Ts>
void fun(Ts... list)
{
  fun_helper(list...);
}

// this is used for recursing through the parameter pack
template <typename T, typename... Ts>
void fun_helper(T first, Ts... rest)
{
  // do thing with first here

  // drop the first element and continue
  fun_helper(rest...);
}

// base case
void fun_helper()
{ }
```

Template Usage & Error checking

Variadic Recursion

```
fun(1, "2", 3.4);
```

Template Usage & Error checking

Variadic Recursion

```
fun(1, "2", 3.4);
```

```
Ts = {int, char const*, double}
```

Template Usage & Error checking

Variadic Recursion

```
fun(1, "2", 3.4);                  Ts = {int, char const*, double}
```

Template Usage & Error checking

Variadic Recursion

```
fun(1, "2", 3.4);

fun_helper(1, "2", 3.4);
```

```
Ts = {int, char const*, double}

First = int, Rest = {char const*, double}
```

Template Usage & Error checking

Variadic Recursion

```
fun(1, "2", 3.4);                    Ts = {int, char const*, double}

fun_helper(1, "2", 3.4);             First = int, Rest = {char const*, double}
```

Template Usage & Error checking

Variadic Recursion

```
fun(1, "2", 3.4);

fun_helper(1, "2", 3.4);

fun_helper("2", 3.4);
```

```
Ts = {int, char const*, double}

First = int, Rest = {char const*, double}

First = char const*, Rest = {double}
```

Template Usage & Error checking

Variadic Recursion

```
fun(1, "2", 3.4);                  Ts = {int, char const*, double}

fun_helper(1, "2", 3.4);           First = int, Rest = {char const*, double}

fun_helper("2", 3.4);              First = char const*, Rest = {double}
```

Template Usage & Error checking

Variadic Recursion

```
fun(1, "2", 3.4);

fun_helper(1, "2", 3.4);

fun_helper("2", 3.4);

fun_helper(3.4);
```

```
Ts = {int, char const*, double}

First = int, Rest = {char const*, double}

First = char const*, Rest = {double}

First = double, Rest = {}
```

Template Usage & Error checking

Variadic Recursion

```
fun(1, "2", 3.4);                 Ts = {int, char const*, double}

fun_helper(1, "2", 3.4);          First = int, Rest = {char const*, double}

fun_helper("2", 3.4);             First = char const*, Rest = {double}

fun_helper(3.4);                  First = double, Rest = {}
```

Template Usage & Error checking

Variadic Recursion

```
fun(1, "2", 3.4);                Ts = {int, char const*, double}

fun_helper(1, "2", 3.4);         First = int, Rest = {char const*, double}

fun_helper("2", 3.4);            First = char const*, Rest = {double}

fun_helper(3.4);                 First = double, Rest = {}

fun_helper();
```

Template Usage & Error checking

Variadic Recursion

Let's use this technique!

Template Usage & Error checking

Variadic Recursion

array.h

```
#include <cstddef> // size_t

template <typename T, size_t N>
class Array
{
public:
  // ...
  template <typename... Ts>
  void set(Ts... list);
  // ...
private:
  void set_helper(size_t i);
  template <typename First, typename... Rest>
  void set_helper(size_t i, First first, Rest... rest);
  // ...
};

#include "array.tcc"
```

Template Usage & Error checking

Variadic Recursion

array.h

```
template <typename T, size_t N>
template <typename... Ts>
void Array<T, N>::set(Ts... list)
{
  set_helper(0, list...);
}

template <typename T, size_t N>
void set_helper(size_t)
{ }

template <typename T, size_t N>
template <typename First, typename... Rest>
void Array<T, N>::set_helper(size_t i, First first, Rest... rest)
{
  data[i] = first;
  set_helper(i+1, rest...);
}
```

Template Usage & Error checking

Variadic Recursion

```
int main()
{
  Array<int, 3> arr;
  arr.set(1,2,3);
}
```

Template Usage & Error checking

Variadic Recursion

```
int main()
{
  Array<int, 3> arr;
  arr.set(1,2,3);
}
```

Nice!

Template Usage & Error checking

Variadic Recursion

```
int main()
{
  Array<int, 3> arr;
  arr.set(1,2,3,4);
}
```

Template Usage & Error checking

Variadic Recursion

```
int main()
{
  Array<int, 3> arr;
  arr.set(1,2,3,4);
}
```

Compiles!

Template Usage & Error checking

Variadic Recursion

```
int main()
{
  Array<int, 3> arr;
  arr.set(1,2,3,4);
}
```

Huh?!

Template Usage & Error checking

Not an error?

- The compiler doesn't perform range checking

Template Usage & Error checking

Not an error?

- The compiler doesn't perform range checking
- So we have to implement it ourselves

Template Usage & Error checking

Not an error?

- The compiler doesn't perform range checking
- So we have to implement it ourselves
- ...But how do we check the number of arguments?

Template Usage & Error checking

`sizeof...`

- `sizeof...` takes a parameter pack;
- return how many elements there are in the give parameter pack;
- will be evaluated during compilation.

Template Usage & Error checking

`sizeof...`

```
template <typename... Ts>
size_t parameter_count(Ts... list)
{
  return sizeof...(list);
}
```

Template Usage & Error checking

But how does this help us?

That's nice and all, but how do we report the error?

Template Usage & Error checking

`static_assert`

- `static_assert` takes two parameters;
- a `bool` which is evaluated during compilation;
- a *message* which is displayed during compilation if the `bool` is false;
- `static_assert`s that fail will halt the compilation.

Template Usage & Error checking

static_assert

```
template <int N>
void check()
{
  static_assert(N > 0, "N must be positive");
}

int main()
{
  check<2>(); // no error
  check<-2>(); // error!
}
```

Template Usage & Error checking

static_assert

```
$ g++ static_assert.cc
static_assert.cc: In instantiation of ‘void check() [with int N = ’-2]:
static_assert.cc:10:13:   required from here
static_assert.cc:4:3: error: static assertion failed: N must be positive
   static_assert(N > 0, "N must be positive");
   ^~~~~~~~~~~~~
```

Template Usage & Error checking

Putting it all together!

array.tcc

```
template <typename T, size_t N>
template <typename... Ts>
void Array<T, N>::set(Ts... list)
{
  static_assert(sizeof...(list) <= N,
                "Too many elements");
  set_helper(0, list...);
}
```

Template Usage & Error checking

That's all folks!

Template Usage & Error checking

That's all folks!

Now wait a minute...

Template Usage & Error checking

What happens if we do this?

```
#include "array.h"

int main()
{
  Array<int, 3> arr;
  arr.set(1,"2",3);
}
```

Template Usage & Error checking

Errors!

```
$ g++ array.cc -std=c++17
In file included from array.h:33:0,
                 from array.cc:1:
array.tcc: In instantiation of ‘void Array<T, N>::set_helper(size_t, First, Rest ...)
[with First = const char*; Rest = {}; T = int; long unsigned int N = 3; size_t = long unsigned int]':
array.tcc:24:15:   recursively required from ‘void Array<T, N>::set_helper(size_t, First, Rest ...)
[with First = int; Rest = {const char*}; T = int; long unsigned int N = 3; size_t = long unsigned int]'
array.tcc:24:15:   required from ‘void Array<T, N>::set_helper(size_t, First, Rest ...)
[with First = int; Rest = {int, const char*}; T = int; long unsigned int N = 3; size_t = long unsigned int]'
array.tcc:16:15:   required from ‘void Array<T, N>::set(Ts ...)
 [with Ts = {int, int, const char*}; T = int; long unsigned int N = 3]'
array.cc:10:23:   required from here
array.tcc:23:13: error: invalid conversion from ‘const char*' to ‘int' [-fpermissive]
     data[i] = head;
     ~~~~~~~~^~~~~~
```

LINKÖPING UNIVERSITY

Type Traits Intro

How do we produce nicer errors?

- The `<type_traits>` header might be helpful

Type Traits Intro

How do we produce nicer errors?

- The <type_traits> header might be helpful
- <type_traits> is used to check various properties about types during compilation

Type Traits Intro

How do we produce nicer errors?

- The `<type_traits>` header might be helpful
- `<type_traits>` is used to check various properties about types during compilation
- Look at cppreference for a complete list

Type Traits Intro

How do we produce nicer errors?

- The `<type_traits>` header might be helpful
- `<type_traits>` is used to check various properties about types during compilation
- Look at cppreference for a complete list
- We will look at `std::is_same`

Type Traits Intro

Simplified implementation of `std::is_same`

```
template <typename T, typename U>
struct is_same
{
  static bool const value{false};
};

template <typename T>
struct is_same<T, T>
{
  static bool const value{true};
};
```

Type Traits Intro

Using `std::is_same`

```
#include <type_traits>
int main()
{
  // true
  bool a{std::is_same<int, int>::value};
  // false
  bool b{std::is_same<int, double>::value};
}
```

Type Traits Intro

Type Traits

- `<type_traits>` was introduced in C++11
- got some nice extensions in C++14
- `is_same_v<T,U>` instead of `is_same<T,U>::value`
- We will talk more about `<type_traits>` later

LINKÖPING UNIVERSITY

Fold Expressions

Fold expressions

- In C++17 *fold expressions* were introduced
- A way to operate on all values in a parameter pack
- Simplifies code significantly, since we do not have to always rely on recursive function templates

Fold Expressions

Fold expression syntax

```
template <typename... Args>
void foo(Args... args)
{
  (args + ...);        // unary right fold
  (... - args);        // unary left fold
  (args + ... + 5); // binary right fold
  (0 * ... * args);  // binary left fold
}
```

Fold Expressions

Fold expression

For `args = {1,2,3,4}`:

```
(args + ...) ==

(... - args) ==

(args + ... + 5) ==

(0 * ... * args) ==
```

Fold Expressions

Fold expression

For `args = {1,2,3,4}`:

```
(args + ...) == 1 + (2 + (3 + 4))

(... - args) ==

(args + ... + 5) ==

(0 * ... * args) ==
```

Fold Expressions

Fold expression

For `args = {1,2,3,4}`:

```
(args + ...) == 1 + (2 + (3 + 4))

(... - args) == ((1 - 2) - 3) - 4

(args + ... + 5) ==

(0 * ... * args) ==
```

Fold Expressions

Fold expression

For `args = {1,2,3,4}`:

```
(args + ...) == 1 + (2 + (3 + 4))

(... - args) == ((1 - 2) - 3) - 4

(args + ... + 5) == 1 + (2 + (3 + (4 + 5)))

(0 * ... * args) ==
```

Fold Expressions

Fold expression

For `args = {1,2,3,4}`:

```
(args + ...) == 1 + (2 + (3 + 4))

(... - args) == ((1 - 2) - 3) - 4

(args + ... + 5) == 1 + (2 + (3 + (4 + 5)))

(0 * ... * args) == (((0 * 1) * 2) * 3) * 4
```

Fold Expressions

Applying fold expressions

```
template <typename T, size_t N>
template <typename... Ts>
void Array<T, N>::set(Ts... list)
{
  // ...
  static_assert((std::is_same_v<T, Ts> && ...),
                "Elements must be of same type");
  set_helper(0, list...);
}
```

Fold Expressions

A little bit better

```
$ g++ array.cc -std=c++17
In file included from array.h:33:0,
                 from array.cc:1:
array.tcc: In instantiation of ‘void Array<T, N>::set(Ts ...)
[with Ts = {int, int, const char*}; T = int; long unsigned int N = 3]':
array.cc:10:23:   required from here
array.tcc:14:5: error: static assertion failed: All types must be the same.
     static_assert((std::is_same_v<T, Ts> && ...),
```

Fold Expressions

Closing Notes

- Fold expressions are very useful when doing generic programming
- we can perform operations over entire ranges of arguments at once
- together with <type_traits> we can do error checking in an easy way

LINKÖPING UNIVERSITY

Namespaces

What is a namespace?

```
namespace NS
{
  void fun();
  int x;
  struct X { };
}
int main()
{
  NS::fun();
  NS::X x{};
  return NS::x;
}
```

Namespaces

What is a namespace?

- Place names inside a named scope;
- good for organizing functionality;
- good example of namespace is `std`.

Namespaces

Importing namespace

```
namespace NS
{
  void fun();
  int x;
  struct X { };
}
using namespace NS;
int main()
{
  fun();
  X x{};
  return x;
}
```

Namespaces

Importing namespace

- `using namespace` will import every name in the namespace into the current name scope;
- very risky, since any name conflict will cause ambiguity;
- use normal `using` declarations instead whenever possible.

Namespaces

Importing parts of the namespace

```
namespace NS
{
  void fun();
  int x;
  struct X { };
}
using NS::fun;
int main()
{
  fun();
  NS::X x{};
  return NS::x;
}
```

Namespaces

Organizing code

- Whenever we declare a name, this will be available in all translation units;
- but often, we want a name to be local to the current translation unit only;
- there are two ways to make this happen `static` or *anonymous namespaces*.

Namespaces

static

```
// foo.h
void foo(int i);

// foo.cc
void foo(int i)
{
  // ...
}
static void foo_helper(int i, int j)
{
  // ...
}
```

Namespaces

`static`

- Briefly in C++98 and C++03 this use of `static` was deprecated;
- but due to compability issues with C this was brought back in C++11;
- `static` declares a function or variable as only visible in the current translation unit.

Namespaces

Anonymous Namespace

foo.cc

```
void foo(int i)
{
  // ...
}
namespace
{
  void foo_helper(int i, int j)
  {
    // ...
  }
}
```

Namespaces

Anonymous Namespace

- A namespace without a name;
- The compiler will generate a name for it and then import it into the current scope;
- Everything inside a anonymous namespace will only be visible inside the current translation unit.

Namespaces

Inner Namespace

```
namespace NS
{
  namespace inner
  {
    void foo();
  }
}
void NS::inner::foo()
{
}
```

Namespaces

Inner Namespace

- Namespaces can be declared inside other namespaces;
- this is useful to further organize your code;
- a technique utilized by the standard library to reduce name conflicts.

Namespace

Inline Namespace

```
namespace NS
{
  inline namespace V2
  {
    void foo();
  }

  namespace V1
  {
    void foo();
  }
}
```

Namespaces

Inline Namespace

- Everything in an inline namespace will automatically be included in the enclosing namespace;
- this feature is used for *symbol versioning*, where we can have various versions available of the same symbols;
- All the "old" versions can be placed in inner namespaces, while the latest version is an inline namespace.

www.liu.se

LINKÖPING UNIVERSITY