TDDD38 - Advanced programming in C++

Templates I

Christoffer Holm

Department of Computer and information science

LINKÖPING UNIVERSITY

LINKÖPING UNIVERSITY

Function Templates

Templates

```
int sum(int (&array)[3])
{
  int result{};
  for (int i{0}; i < 3; ++i)
  {
    result += array[i];
  }
  return result;
}
```

Function Templates

Templates

```
int sum(int (&array)[3])
{
  int result{};
  for (int i{0}; i < 3; ++i)
  {
    result += array[i];
  }
  return result;
}
```

```
double sum(double (&array)[3])
{
  double result{};
  for (int i{0}; i < 3; ++i)
  {
    result += array[i];
  }
  return result;
}
```

Function Templates

Templates

```
int sum(int (&array)[3])
{
  int result{};
  for (int i{0}; i < 3; ++i)
  {
    result += array[i];
  }
  return result;
}
```

```
string sum(string (&array)[3])
{
  string result{};
  for (int i{0}; i < 3; ++i)
  {
    result += array[i];
  }
  return result;
}
```

```
double sum(double (&array)[3])
{
  double result{};
  for (int i{0}; i < 3; ++i)
  {
    result += array[i];
  }
  return result;
}
```

Function Templates

Templates

```
int sum(int (&array)[3])
{
  int result{};
  for (int i{0}; i < 3; ++i)
  {
    result += array[i];
  }
  return result;
}
```

```
double sum(double (&array)[3])
{
  double result{};
  for (int i{0}; i < 3; ++i)
  {
    result += array[i];
  }
  return result;
}
```

```
string sum(string (&array)[3])
{
  string result{};
  for (int i{0}; i < 3; ++i)
  {
    result += array[i];
  }
  return result;
}
```

```
int main()
{
  int    arr1[3] { 5, 5, 5 };
  double arr2[3] { 1.05, 1.05, 1.04 };
  string arr3[3] { "h", "i", "!" };
  cout << sum(arr1) << endl;
  cout << sum(arr2) << endl;
  cout << sum(arr3) << endl;
}
```

Function Templates

Templates

```
template <typename T>
T sum(T (&array)[3])
{
  T result{};
  for (int i{0}; i < 3; ++i)
  {
    result += array[i];
  }
  return result;
}
```

Function Templates

Templates

```
template <typename T>
T sum(T (&array)[3])
{
  T result{};
  for (int i{0}; i < 3; ++i)
  {
    result += array[i];
  }
  return result;
}
```

```
int main()
{
  int    arr1[3] { 5, 5, 5 };
  double arr2[3] { 1.05, 1.05, 1.04 };
  string arr3[3] { "h", "i", "!" };
  cout << sum(arr1) << endl;
  cout << sum(arr2) << endl;
  cout << sum(arr3) << endl;
}
```

Function Templates

Templates

- Generalizes functions with arbitrary data types;
- the data types are filled in during *compilation*;
- generate functions when they are used in the code.

Function Templates

Templates

- Template-parameters *are not* dynamic types;
- they are just placeholders inside a template;
- the user will pass in types into the function template;
- when this occurs, a function is generated;
- this is called *template instantiation*.

Function Templates

Template Instantiation

```
template <typename T>
T sum(T (&array)[3])
{
  T result{};
  for (int i{0}; i < 3; ++i)
  {
    result += array[i];
  }
  return result;
}
```

```
int main()
{
  int    arr1[3] { 1, 2, 3 };
  double arr2[3] { 4.5, 6.7, 8.9 };
  // explicitly instantiate sum
  cout << sum<int>(arr1) << endl;
  // let the compiler instantiate sum
  cout << sum(arr2) << endl;
}
```

Function Templates

Default Parameters

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

```
int main()
{
  cout << identity() << endl;
  cout << identity<double>(3.0) << endl;
  cout << identity<string>() << endl;
}
```

Function Templates

Template Argument Deduction

- To instantiate a function template every template argument must be known;
- however: the user does not have to specify all template arguments;
- whenever possible the compiler will deduce the arguments that the user left out;
- this is called *template argument deduction*.

Function Templates

Template Argument Deduction

```
int main()
{
  int arr1[3] { 1, 2, 3 };
  cout << sum(arr1) << endl;
}
```

Function Templates

Template Argument Deduction

```
int main()
{
  int arr1[3] { 1, 2, 3 };
  cout << sum(arr1) << endl;
}
```

Function Templates

Template Argument Deduction

```
int main()
{
  int arr1[3] { 1, 2, 3 };
  cout << sum(arr1) << endl;
}
```

Function Templates

Template Argument Deduction

```
int main()
{
  int arr1[3] { 1, 2, 3 };
  cout << sum<int>(arr1) << endl;
}
```

Function Templates

Template Argument Deduction

```
template <typename T1, typename T2>
void print(T1 a, T2 b)
{
  cout << a << ' ' << b << endl;
}
```

```
int main()
{
  string a{"val ="};
  print(a, 5);
}
```

Function Templates

Template Argument Deduction

```
template <typename T1, typename T2>
void print(T1 a, T2 b)
{
  cout << a << ' ' << b << endl;
}
```

```
int main()
{
  string a{"val ="};
  print<string, int>(a, 5);
}
```

Function Templates

Return Types

```cpp
template <typename Ret, typename T1, typename T2>
Ret max(T1 const& a, T2 const& b)
{
  if (a > b)
    return a;
  return b;
}
```

Function Templates

Return Types

```
template <typename Ret, typename T1, typename T2>
Ret max(T1 const&, T2 const&);
int main()
{
  // ill-formed, cannot deduce 'Ret'
  cout << max(5, 6.0)
       << endl;
}
```

Function Templates

Return Types

```cpp
template <typename Ret, typename T1, typename T2>
Ret max(T1 const&, T2 const&);
int main()
{
  // works, but tedious
  cout << max<double, int, double>(5, 6.0)
       << endl;
}
```

Function Templates

Return Types

```
template <typename Ret, typename T1, typename T2>
Ret max(T1 const&, T2 const&);
int main()
{
  // works and is nice!
  cout << max<double>(5, 6.0)
       << endl;
}
```

Function Templates

Return Types

- The user have to specify `Ret` since the compiler is unable to deduce the return type;
- however the compiler will deduce `T1` and `T2` from the arguments passed to the function.

Function Templates

Going back to our example

```
template <typename T>
T sum(T (&array)[3])
{
  T result{};
  for (int i{0}; i < 3; ++i)
  {
    result += array[i];
  }
  return result;
}
```

Function Templates

Going back to our example

```cpp
template <typename T>
T sum(T (&array)[3])
{
  T result{};
  for (int i{0}; i < 3; ++i)
  {
    result += array[i];
  }
  return result;
}
```

```cpp
string sum(string (&array)[3])
{
  string result{};
  for (int i{0}; i < 3; ++i)
  {
    result += array[i] + " ";
  }
  return result;
}
```

Function Templates

Overload Resolution

```
int main()
{
  int    arr1[3] { 1, 2, 3 };
  string arr2[3] { "templates",
                   "are", "fun" };
  cout << sum(arr1) << endl;
  cout << sum(arr2) << endl;
}
```

```
// Candidates

template <typename T> T sum(T (&)[3]);

string sum(string (&)[3]);
```

Function Templates

Overload Resolution

```
int main()
{
  int    arr1[3] { 1, 2, 3 };
  string arr2[3] { "templates",
                   "are", "fun" };
  cout << sum(arr1) << endl;
  cout << sum(arr2) << endl;
}
```

```
// Candidates

template <typename T> T sum(T (&)[3]);

string sum(string (&)[3]);
```

Function Templates

Overload Resolution

```
int main()
{
  int    arr1[3] { 1, 2, 3 };
  string arr2[3] { "templates",
                   "are", "fun" };
  cout << sum(arr1) << endl;
  cout << sum(arr2) << endl;
}
```

```
// Candidates

template <typename T> T sum(T (&)[3]);

string sum(string (&)[3]);
```

Function Templates

Overload Resolution

```
int main()
{
  int    arr1[3] { 1, 2, 3 };
  string arr2[3] { "templates",
                   "are", "fun" };
  cout << sum(arr1) << endl;
  cout << sum(arr2) << endl;
}
```

```
// Candidates

template <typename T> T sum(T (&)[3]);

string sum(string (&)[3]);
```

Function Templates

Overload Resolution

```
int main()
{
  int    arr1[3] { 1, 2, 3 };
  string arr2[3] { "templates",
                   "are", "fun" };
  cout << sum(arr1) << endl;
  cout << sum(arr2) << endl;
}
```

```
// Candidates

template <typename T> T sum(T (&)[3]);

string sum(string (&)[3]);
```

Function Templates

Overload Resolution

If a function call is performed;

1. *name lookup* to find candidate functions;
2. *overload resolution* decides which function to call.

Function Templates

Overload Resolution

If a function call is performed;

1. *name lookup* to find candidate functions;
    - *Qualified name lookup*
    - *Unqualified name lookup*
    - *Argument dependent lookup*
2. *overload resolution* decides which function to call.

Function Templates

Name Lookup

- *Qualified name lookup*
- *Unqualified name lookup*
- *Argument dependent lookup*

Function Templates

Name Lookup

- *Qualified name lookup*
  - *Qualified names* are those names that appear to the right of `::` operators;
  - A qualified name is a class member, namespace member, inner namespace or enum;
  - The name to the left of `::` must be found before lookup can be performed on the right-hand-side.
- *Unqualified name lookup*
- *Argument dependent lookup*

Function Templates

Name Lookup

- *Qualified name lookup*
- *Unqualified name lookup*
  - *Unqualified names* are all names that are not qualified;
  - Will look in the current scope before traversing *outwards* to the enclosing scope;
  - This means that names can be overriden inside inner scopes.
- *Argument dependent lookup*

Function Templates

Name Lookup

- *Qualified name lookup*
- *Unqualified name lookup*
- *Argument dependent lookup*
    - A special case of unqualified name lookup;
    - If the unqualified name cannot be found with normal means;
    - Look inside the scope where the type of the arguments is defined;
    - Example: `std::cout << "hello";`
    - Often denoted as *ADL*.

Function Templates

Overload Resolution

If a function call is performed;

1. *name lookup* to find candidate functions;
2. *overload resolution* decides which function to call.
    1. exact matches
    2. function templates
    3. argument conversions
    4. overload resolution fails

Function Templates

Overload Resolution

1. exact matches
   - If a non-template candidate function *exactly* matches the argument types;
   - This overload rule makes it possible to overload functions.
2. function templates
3. argument conversions
4. overload resolution fails

Function Templates

Overload Resolution

1. exact matches
2. function templates
   - If a function template can be instantiated to exactly match the argument types;
   - Will not always work, since some arguments might not be templated.
3. argument conversions
4. overload resolution fails

Function Templates

Overload Resolution

1. exact matches
2. function templates
3. argument conversions
    - type conversion can be applied to the arguments;
    - if more than one function is considered, then the *best match* is selected;
    - if no unique best match is available, then none is selected;
    - this step will also consider template functions.
4. overload resolution fails

Function Templates

Overload Resolution

1. exact matches
2. function templates
3. argument conversions
4. overload resolution fails
   - Either no appropriate function could be found;
   - or there where multiple equally fitting candidate functions.

Function Templates

Function Specialization

```
template <>
string sum<string>(string (&array)[3])
{
  string result{};
  for (int i{0}; i < 3; ++i)
  {
    result += array[i] + " ";
  }
  return result;
}
```

Function Templates

Function Specialization

primary template

```
template <typename T>
T sum(T (&array)[3])
{
  T result{};
  for (int i{0}; i < 3; ++i)
  {
    result += array[i];
  }
  return result;
}
```

explicit template specialization

```
template <>
string sum<string>(string (&array)[3])
{
  string result{};
  for (int i{0}; i < 3; ++i)
  {
    result += array[i] + " ";
  }
  return result;
}
```

Function Templates

Function Specialization

- Override specific instantiations of the primary template;
- specializations are not considered in overload resolutions;
- however, the specialization will be used if the primary template is considered;
- *always* prefer normal functions whenever possible, since they will be prioritized over all function templates.

Function Templates

Function Specialization

```
template <typename T>
T fun();

// won't work, ambigious
int fun();

// will work, since specializations override
// the primary template
template <> int fun<int>();
```

Function Templates

What will happen? Why?

```
template <typename T>
void print(T) { cout << "1"; }

template <>
void print<int>(int) { cout << "2"; }

void print(int&&) { cout << "3"; }

int main()
{
    int x{};
    print(1.0);
    print(1);
    print(x);
}
```

LINKÖPING UNIVERSITY

Nontype Template Parameters

Generalize our program

```
template <typename T>
T sum(T (&array)[3])
{
  T result{};
  for (int i{0}; i < 3; ++i)
  {
    result += array[i];
  }
  return result;
}
```

Nontype Template Parameters

Generalize our program

```
template <typename T, unsigned N>
T sum(T (&array)[N])
{
  T result{};
  for (int i{0}; i < N; ++i)
  {
    result += array[i];
  }
  return result;
}
```

Nontype Template Parameters

Restrictions

- All values passed into nontype parameters must be known during compilation;

Nontype Template Parameters

Restrictions

- All values passed into nontype parameters must be known during compilation;
- the type of the nontype parameter is limited;
    - *integral types*
    - *enum types*
    - *pointers*
    - *lvalue references*

Nontype Template Parameters

Restrictions

- All values passed into nontype parameters must be known during compilation;
- the type of the nontype parameter is limited;
  - *integral types*
  - *enum types*
  - *pointers*
  - *lvalue references*
- all nontype parameters, except for references, are *prvalues*.

Nontype Template Parameters

Restrictions

- All values passed into nontype parameters must be known during compilation;
- the type of the nontype parameter is limited;
    - *integral types*
    - *enum types*
    - *pointers*
    - *lvalue references*
- all nontype parameters, except for references, are *prvalues*.
    - are not modifiable
    - does not have an address
    - cannot be bound to lvalue references

Nontype Template Parameters

Restrictions

```
template <int x, int& y>
void foo()
{
    ++x; // ill-formed
    ++y; // OK
    &x; // ill-formed
    &y; // OK
    int& z = x; // ill-formed
    int& w = y; // OK
}

int x{};
int main()
{
    foo<5, x>();
}
```

Nontype Template Parameters

Fibonacci Example

```
template <int N = 2>
int fib()
{
    return fib<N-2>() + fib<N-1>();
}
template <> int fib<0>() { return 1; }
template <> int fib<1>() { return 1; }

int main()
{
    cout << fib<6>() << endl;
    cout << fib() << endl;
}
```

Nontype Template Parameters

Fibonacci Example

- It is possible to set default values for template parameters;

Nontype Template Parameters

Fibonacci Example

- It is possible to set default values for template parameters;
- note that this use of explicit template specializations is required;

Nontype Template Parameters

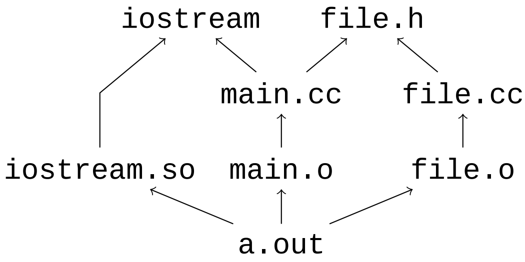
Fibonacci Example

- It is possible to set default values for template parameters;
- note that this use of explicit template specializations is required;
- using templates to perform calculations during compile-time.

LINKÖPING UNIVERSITY

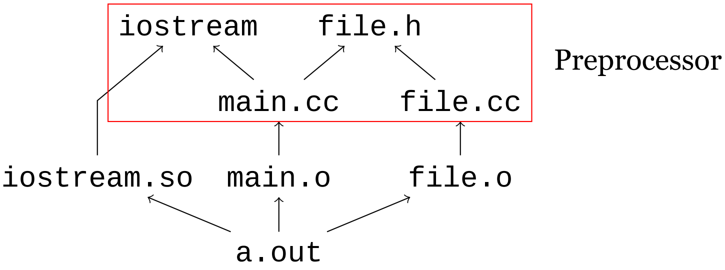
Compilation and Linking

The compilation process



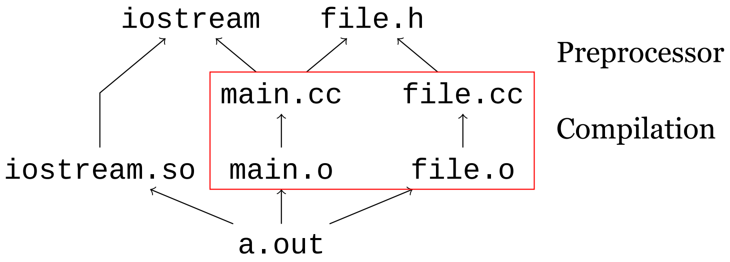
Compilation and Linking

The compilation process



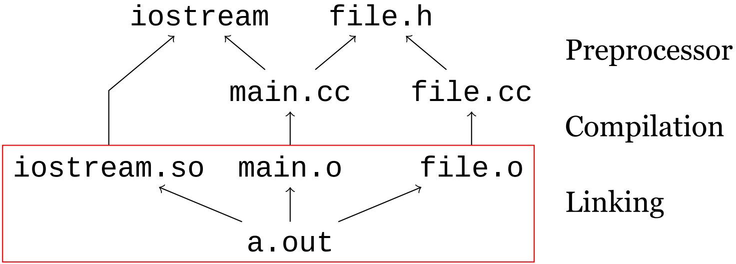
Compilation and Linking

The compilation process



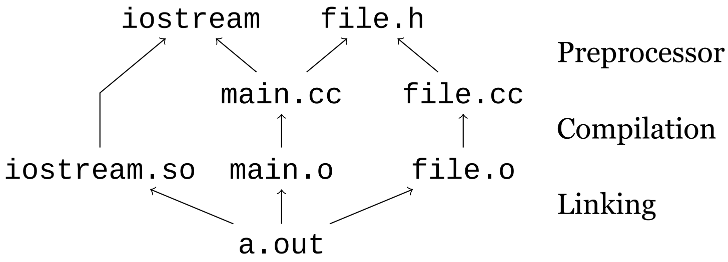
Compilation and Linking

The compilation process



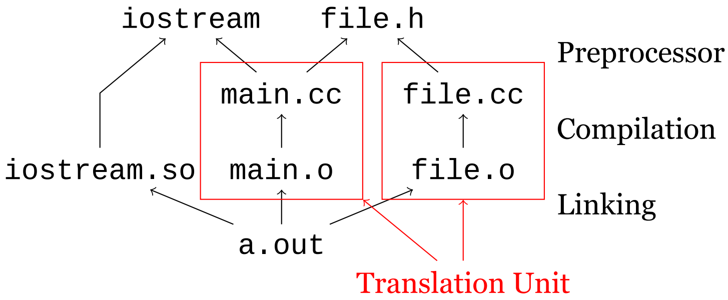
Compilation and Linking

The compilation process



Compilation and Linking

The compilation process



Compilation and Linking

Compiling Templates

- When the compiler instantiates a template, the entire definition of the template must be known;
- the compiler will compile each cc-file in individual *translation units*;
- it is during the *compilation* step that the compiler instantiate templates;
- because of this, the entire definition of the template must be known inside each translation unit, meaning it is required that the templates are completely defined in the header or implementation file directly.

Compilation and Linking

A way to structure your files

```
// foo.h
#ifndef FOO_H_
#define FOO_H_

template <typename T>
T foo(T);

// do NOT compile foo.tcc
// it will be handled
// when foo.h is included
#include "foo.tcc"
#endif FOO_H_
```

```
// foo.tcc
template <typename T>
T foo(T t)
{
  return t;
}
```

```
#include "foo.h"
int main()
{
  cout << foo(5) << endl;
}
```

Compilation and Linking

Compiling specializations

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

```
template <>
int foo()
{
  return 1;
}

// must be instantiated
// after the declaration
// of the specialization.
int n{foo<int>()};
```

Compilation and Linking

Compiling specializations

- Specializations must be declared before the first explicit instantiation of that specialization of the function template;
- both the definition and the declaration of the specialization must be known;
- the compiler can have a hard time to detect these kinds of errors;
- always make sure that all your function templates are declared before you use them.

Compilation and Linking

Helpful limerick from the standard

When writing a specialization,
be careful about its location;
or to make it compile
will be such a trial
as to kindle its self-immolation.

[temp.expl.spec]-§7

Constexpr and auto

The `auto` keyword in declarations

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

```
auto foo(auto t)
{
  return t;
}
```

Constexpr and auto

`auto` parameters

- Whenever the compiler finds a function declaration where one or more parameters are declared as `auto`, a function template will be generated;
- each distinct parameter declared as auto will result in one disting template parameter;
- this is just syntactic sugar, but it will significantly increase the readability of code.

Constexpr and auto

auto as return type

```
auto add(int a, double b)
{
  return a + b;
}
```

```
auto add(auto a, auto b)
{
  return a + b;
}
```

Constexpr and auto

`auto` as return type

- If the return type is declared as `auto`, the compiler will try to deduce the return type from the `return` statements inside the function;
- if there are multiple `return` statements in the body of the function, the types of the expressions returned must be *exactly* the same for each `return` statement;
- this behaviour work for both normal functions and function templates.

Constexpr and auto

Example when `auto` return type fails

```
auto foo(int t)
{
  if (t < 0)
  {
    return false;
  }
  return 1;
}
```

Constexpr and auto

`constexpr`

- One of the big advantages of templates are that they are evaluated during compile-time rather than run-time;
- this makes it possible to use templates for calculations during compile-time;
- however, most problems are solved differently when dealing with compile-time;
- in C++11 the `constexpr` keyword was introduced, which is meant to bridge this gap.

Constexpr and auto

constexpr

```
constexpr int fib(int N = 2)
{
    if (N <= 1) return 1;
    return fib(N-2) + fib(N-1);
}

int main()
{
    cout << fib(6) << endl;
    cout << fib() << endl;
}
```

www.liu.se

LINKÖPING
UNIVERSITY