TDDD38/726G82 - Advanced programming in C++

Templates I

Christoffer Holm

Department of Computer and information science

LINKÖPING UNIVERSITY

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

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

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

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

Function Templates

Template Argument Deduction

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

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

Function Templates

Template Argument Deduction

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

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

Function Templates

Template Argument Deduction

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

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

Function Templates

Template Argument Deduction

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

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

Function Templates

Template Argument Deduction

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

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;
}
```

```
using namespace std::literals;
int main()
{
  print("val = ", 5);
}
```

Function Templates

Template Argument Deduction

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

```
using namespace std::literals;
int main()
{
  print<char const*, int>("val = ", 5);
}
```

Function Templates

Template Argument Deduction

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

```
using namespace std::literals;
int main()
{
  print("val = "s, 5);
}
```

Function Templates

Template Argument Deduction

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

```
using namespace std::literals;
int main()
{
  print<string, int>("val = "s, 5);
}
```

Function Templates

Return Types

```
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

```
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

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

```
template <typename T>
T sum(T (&array)[3])
{
  T 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;
}
```

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

Name Lookup

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

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

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

```
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

```
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

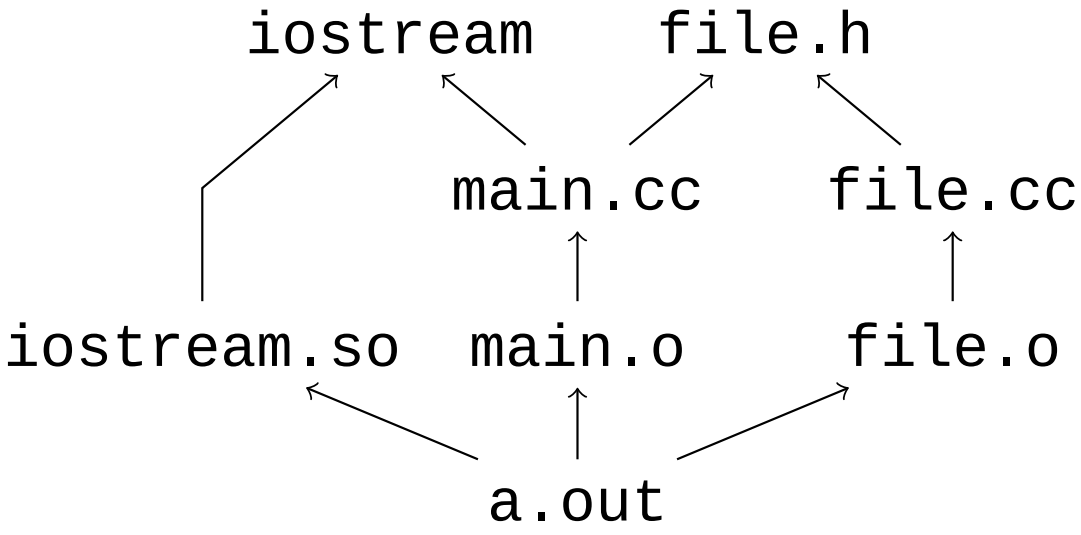
```
template <int N = 2>
int fib()
{
    return fib<N-2>() + fib<N-1>();
}
template <> int fib<0>() { return 1; }
template <> int fib<1>() { return 1; }

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

LINKÖPING UNIVERSITY

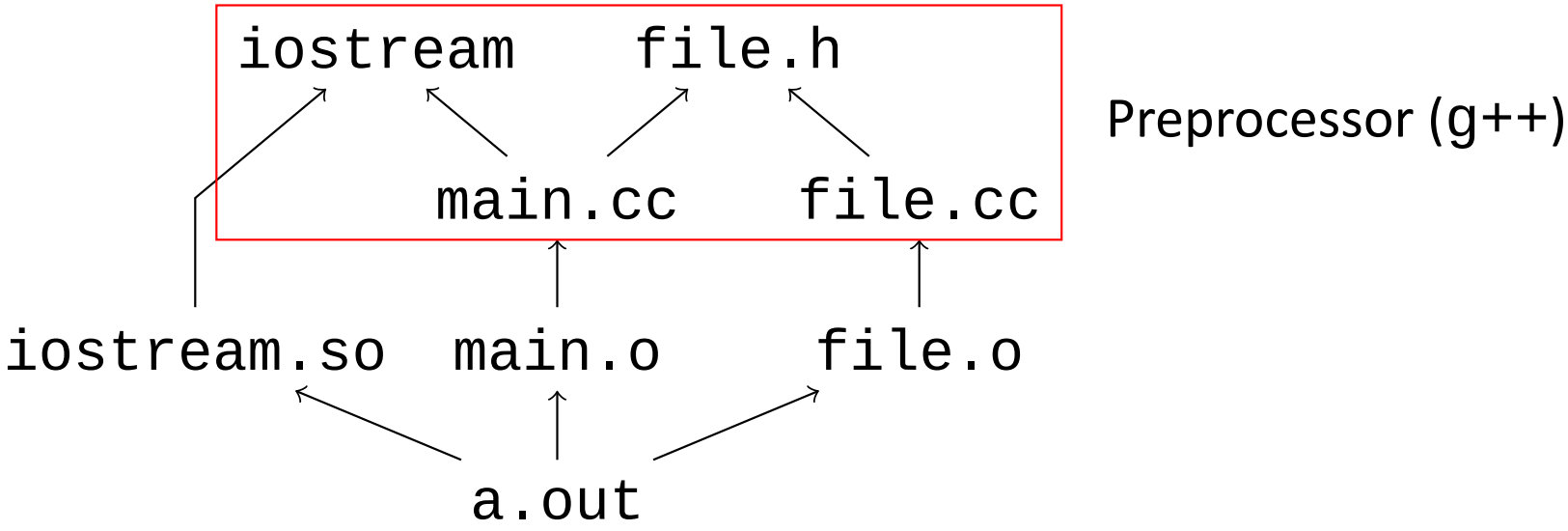
Compilation and Linking

The compilation process



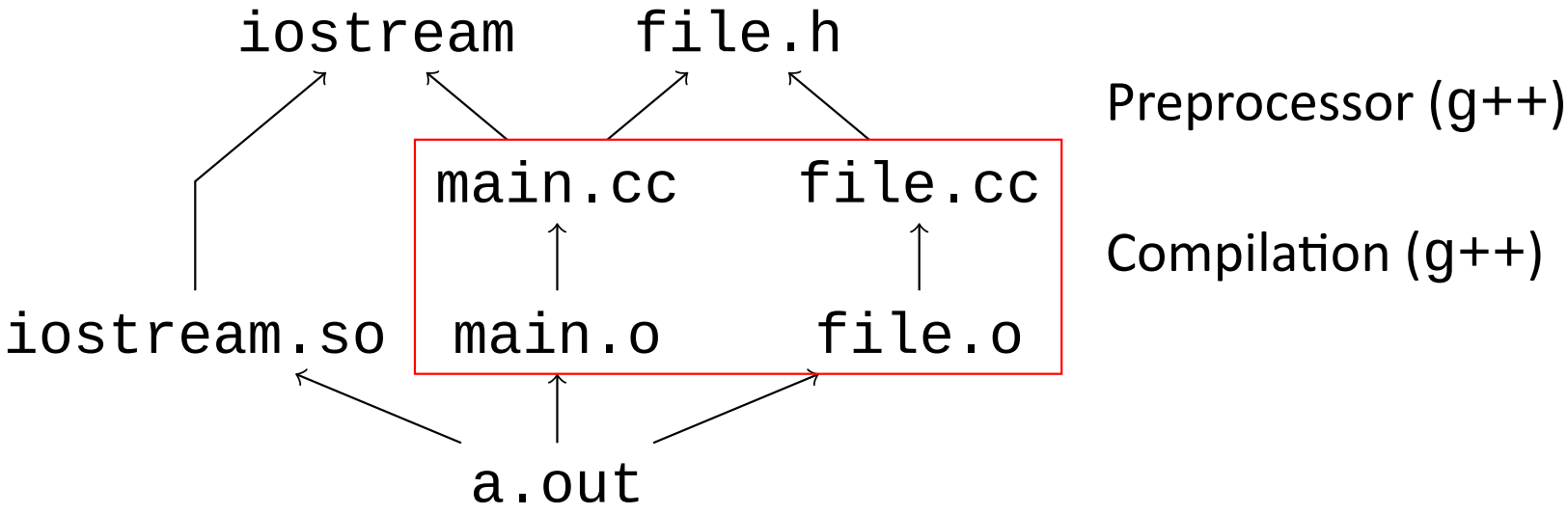
Compilation and Linking

The compilation process



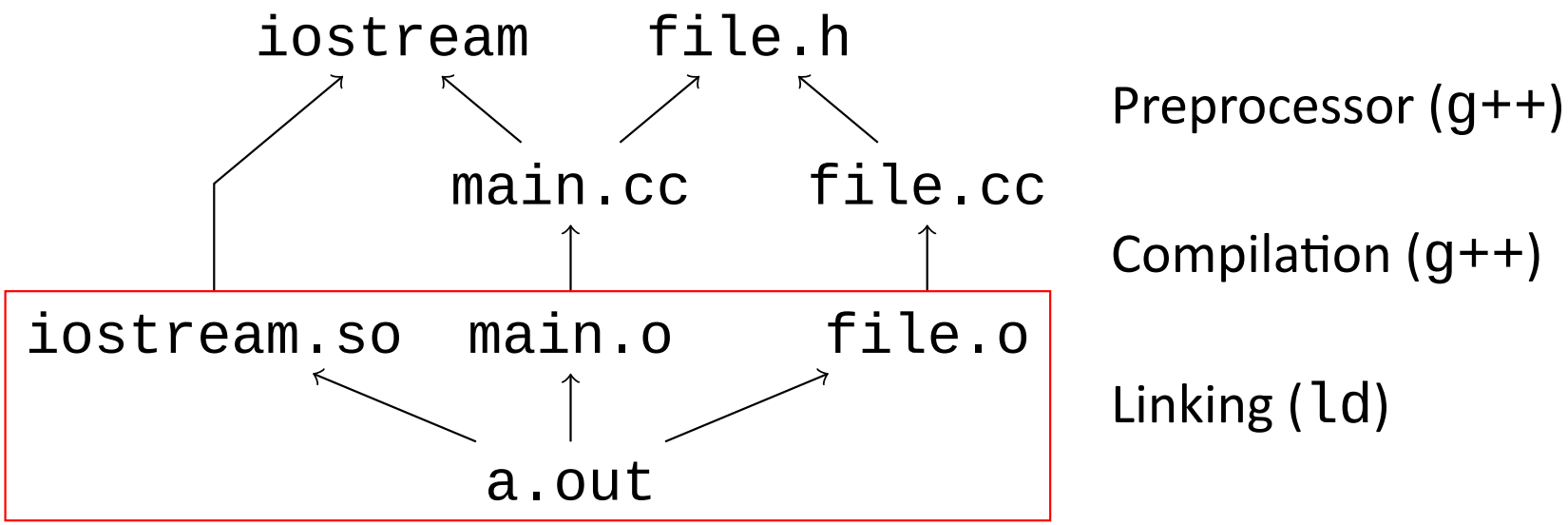
Compilation and Linking

The compilation process



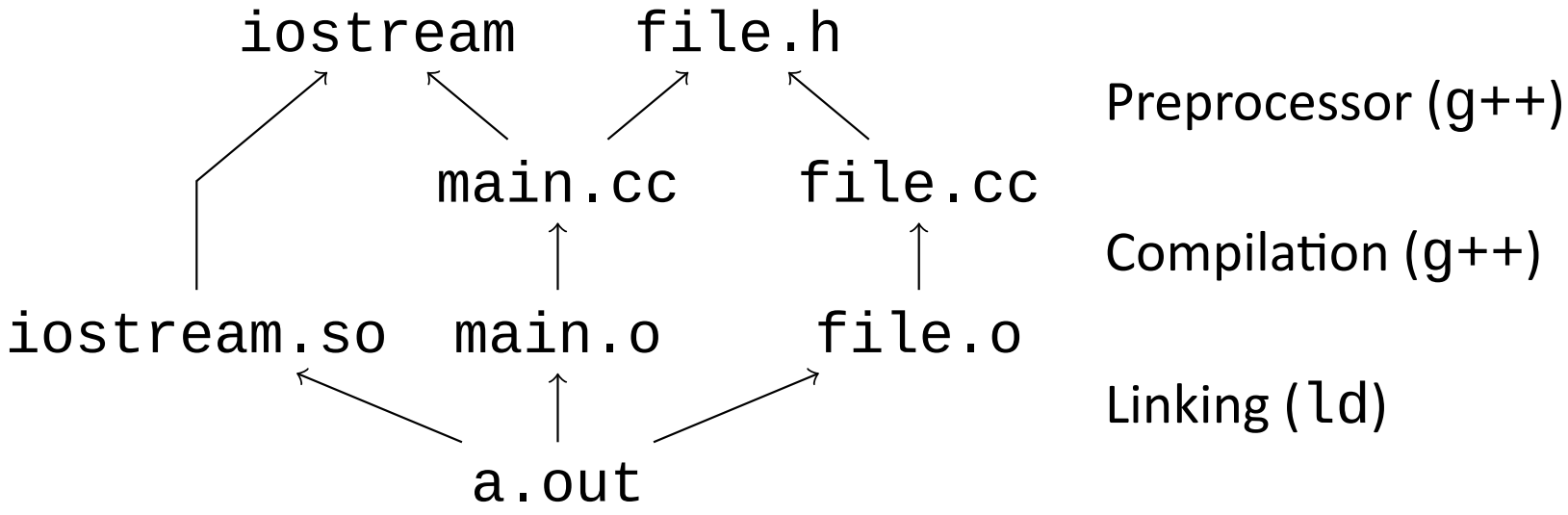
Compilation and Linking

The compilation process



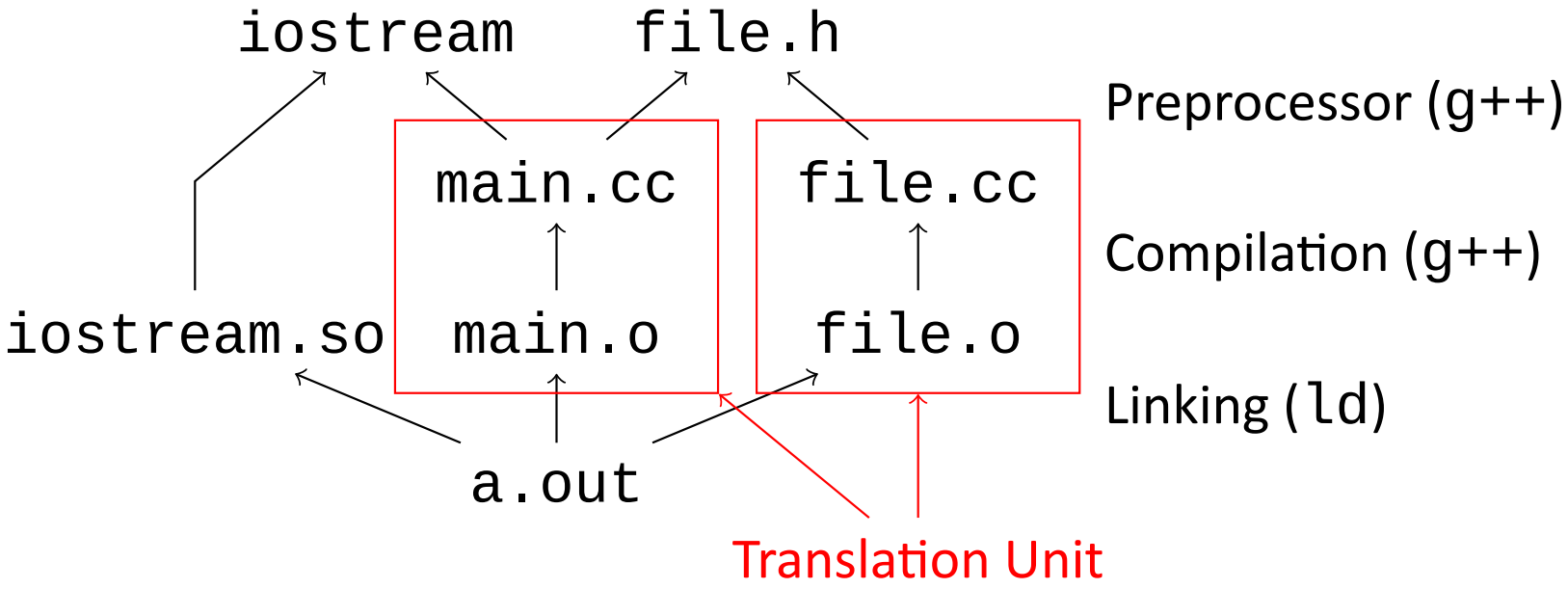
Compilation and Linking

The compilation process



Compilation and Linking

The compilation process



Compilation and Linking

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

What will be printed? Why?

```
template <typename T>
void fun(T) { cout << 1 << endl; }

template <>
void fun(int*) { cout << 2 << endl; }

template <typename T>
void fun(T*) { cout << 3 << endl; }

int main()
{
  int* x{};
  double* y{};

  fun(1);
  fun(x);
  fun(y);
}
```

Compilation and Linking

What will be printed? Why?

```
template <typename T>
void fun(T) { cout << 1 << endl; }

template <typename T>
void fun(T*) { cout << 3 << endl; }

template <>
void fun(int*) { cout << 2 << endl; }

int main()
{
  int* x{};
  double* y{};

  fun(1);
  fun(x);
  fun(y);
}
```

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

LINKÖPING UNIVERSITY

Constexpr and auto

auto

```
auto s { "hello" }; // char const*
auto n { 5 }; // int
auto x; // NOT allowed

n = s; // NOT allowed
```

Constexpr and auto

auto

```
auto s { "hello"s }; // std::string
auto n { 5 }; // int
auto x; // NOT allowed

n = s; // NOT allowed
```

Constexpr and auto

`auto` as return type

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

```
template <typename T1,
          typename T2>
auto add(T1 a, T2 b)
{
  return a + b;
}
```

Constexpr and auto

Example when `auto` return type fails

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

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;
}
```

Constexpr and auto

C++20: `auto` parameters

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

```
template <typename T1,
          typename T2>
auto foo(T1 a, T2 b)
{
  return a + b;
}
```

www.liu.se

LINKÖPING
UNIVERSITY