Intro to C++
Template Meta Programming

Architecture Engine Club Summer 2023

ADVANCED TOPICS

C++ TEMPLATES

Overview

- Type Traits
- `if constexpr`
- Static Assert
- Variadic Arguments
- Fold Expressions
- Explicit Instantiation

Type Traits

TYPE TRAITS

Definition

“Classes to obtain characteristics of types in the form of compile-time constant values.”

- cplusplus.com

“Compile-time constants that give information about the properties of their type arguments”

- Microsoft

TYPE TRAITS

Definition TL;DR

Gives information about a type at compile-time

TYPE TRAITS

Example

```
std::is_floating_point<T>::value;
std::is_same<T, V>::value;
std::is_arithmetic<T>::value;
```

TYPE TRAITS

Example

Short-hand for easier typing and readability

```
std::is_floating_point_v<T>;
std::is_same_v<T, V>;
std::is_arithmetic_v<T>;
```

TYPE TRAITS

Question

Would this work the way we want?

```
template<typename T>
bool isInt(T&& val)
{
    return std::is_same_v<int, T>;
}
const int& X = 5;
isInt(X);  // True or False?
```

Note: T&& is a forwarding reference for advanced use cases. Only use it if you know what you are doing.

TYPE TRAITS

Problem

T is actually const int&

```
template<typename T>
bool isInt(T&& val)
{
    return std::is_same_v<int, T>;
}
const int& X = 5;
isInt(X); // False
```

Note: T&& is a forwarding reference for advanced use cases. Only use it if you know what you are doing.

TYPE TRAITS

Solution

Remove unwanted contain `const`, `volatile` or & qualifiers

```
using Type = std::remove_cvref<T>::type;

std::is_floating_point_v<Type>;
std::is_same_v<Type, V>;
std::is_arithmetic_v<Type>;
```

TYPE TRAITS

Solution

Of course, it also has a short-hand

```
using Type = std::remove_cvref_t<T>;

std::is_floating_point_v<Type>;
std::is_same_v<Type, V>;
std::is_arithmetic_v<Type>;
```

TYPE TRAITS

Solved Example

Now it works as we want it to

```
template<typename T>
bool isInt(T&& val)
{
    using Type = std::remove_cvref_t<T>;
    return std::is_same_v<int, Type>;
}
const int& X = 5;
isInt(X); // True
```

Note: T&& is a forwarding reference for advanced use cases. Only use it if you know what you are doing.

if constexpr

IF CONSTEXPR

The Problem

What happens if T is a float?

```
template<typename T>
bool equals(T a, T b)
{
    return a == b;
}
```

IF CONSTEXPR

The Problem

What happens if T is a float?

Non-Floats

```
a == b
```

Floats

```
abs(a - b) < EPSILON
```

```
template<typename T>
bool equals(T& a, T b)
{
    return a == b;
}
```

IF CONSTEXPR

Definition

Let’s you choose what code is included in your function template instantiation

```
if constexpr (/* constexpr expression */)
    // Do X
else
    // Do Y
```

IF CONSTEXPR

The Solution

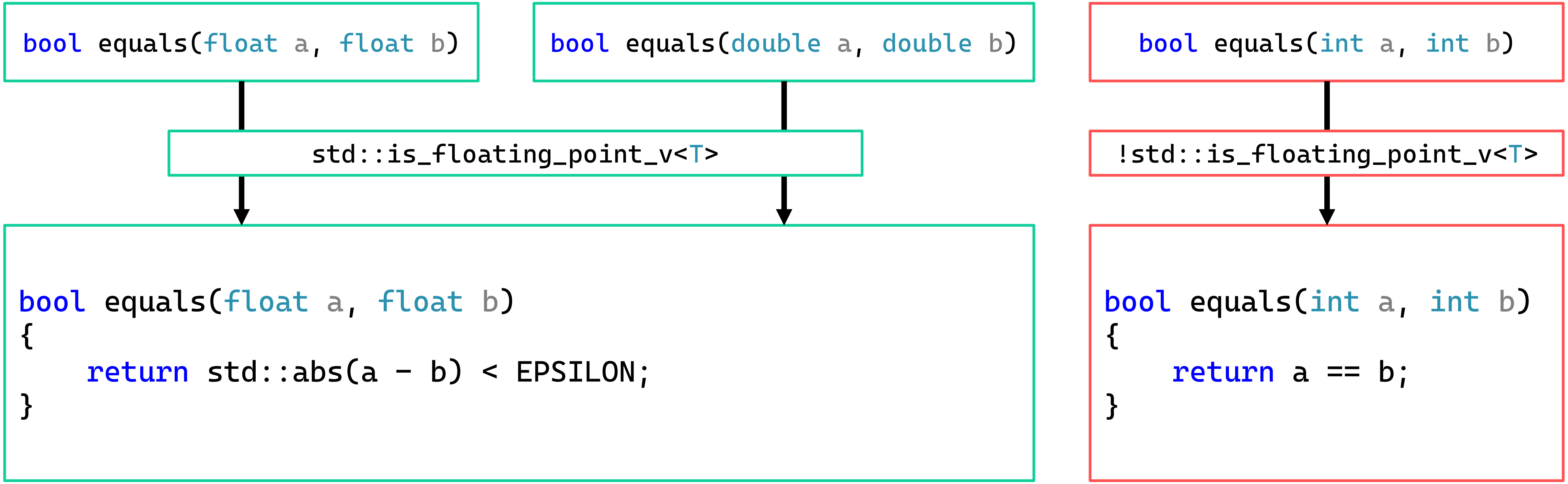
Use `if constexpr` with type traits to branch our code at compile-time

```
template<typename T>
bool equals(T a, T b)
{
    if constexpr (std::is_floating_point_v<T>)
        return std::abs(a - b) < EPSILON;
    else
        return a == b;
}
```

IF CONSTEXPR

The Solution

Use `if constexpr` with type traits to branch our code at compile-time



```
bool equals(float a, float b)
```

```
bool equals(double a, double b)
```

```
bool equals(int a, int b)
```

```
std::is_floating_point_v<T>
```

```
!std::is_floating_point_v<T>
```

```
bool equals(float a, float b)
{
    return std::abs(a - b) < EPSILON;
}
```

```
bool equals(int a, int b)
{
    return a == b;
}
```

Static Assert

STATIC ASSERT

The Problem

Would this code work?

```
template<typename T>
float sumXChannel(const T* begin, const T* end)
{
    T sum = T();
    for (const T* iter = begin; iter != end; ++iter)
    {
        sum.x += iter->x;
    }
    return sum;
}

const Vec3 DATA[SIZE] = { ... };
sumXChannel(&DATA[0], &DATA[SIZE]);
```

STATIC ASSERT

The Problem

What about this?

```
template<typename T>
float sumXChannel(const T* begin, const T* end)
{
    T sum = T();
    for (const T* iter = begin; iter != end; ++iter)
    {
        sum.x += iter->x;
    }
    return sum;
}

const float DATA[SIZE] = { ... };
sumXChannel(&DATA[0], &DATA[SIZE]);
```

STATIC ASSERT

The Problem

What about this?

```
template<typename T>
T sumXChannel(const T* begin, const T* end)
{
    T sum = T();
    for (const T* iter = begin; iter != end; ++iter)
    {
        sum.x += iter->x;
    }
    return sum;
}

const float DATA[SIZE] = { ... };
sumXChannel(&DATA[0], &DATA[SIZE]);
```

```
1>D:\CPPTemplates.cpp(21,13): error C2228: left of '.x' must have class/struct/union
1>D:\CPPTemplates.cpp(21,13): message : type is 'float'
1>D:\CPPTemplates.cpp(48,29): message : see reference to function template
instantiation 'T sumXChannel<float>(const T *,const T *)' being compiled
1>        with
1>        [
1>            T=float
1>        ]
```

STATIC ASSERT

The Solution

```
template<typename T>
T sumXChannel(const T* begin, const T* end)
{
    static_assert(std::is_same_v<T, Vec3>, "Only Vec3 is supported.");
    T sum = T();
    for (const T* iter = begin; iter != end; ++iter)
    {
        sum.x += iter->x;
    }
    return sum;
}

const float DATA[SIZE] = { ... };
sumXChannel(&DATA[0], &DATA[SIZE]);
```

STATIC ASSERT

The Solution

```
template<typename T>
T sumXChannel(const T* begin, const T* end)
{
    static_assert(std::is_same_v<T, Vec3>, "Only Vec3 is supported.");
    T sum = T();
    for (const T* iter = begin; iter != end; ++iter)
    {
        sum.x += iter->x;
    }
    return sum;
}

const float DATA[SIZE] = { ... };
sumXChannel(&DATA[0], &DATA[SIZE]);
```

```
1>D:\CPPTemplates.cpp(16,19): error C2338: static_assert failed: 'Only Vec3 is supported'
1>D:\CPPTemplates.cpp(48,29): message : see reference to function template instantiation
'T sumXChannel<float>(const T *,const T *)' being compiled
1>        with
1>        [
1>            T=float
1>        ]
```

Variadic Arguments

VARIADIC ARGUMENTS

The Problem

Can we do this in a single call?

```
void subscribe(void* sub, MSG_TYPE m)
{
    subscribers[m].push_back(sub);
}

subscribe(this, MSG_1);
subscribe(this, MSG_2);
subscribe(this, MSG_3);
subscribe(this, MSG_4);
```

VARIADIC ARGUMENTS

The Problem

Ideally, we want something like this right?

```
void subscribe(void* sub, ???)
{
    ???
}

subscribe(this, MSG_1, MSG_2, MSG_3, MSG_4);
```

VARIADIC ARGUMENTS

Definition

Allows you to have multiple variables without using `va_args`

```
template<typename ... Args>
void foo(Args ... args)
{
}
```

VARIADIC ARGUMENTS

Parameter Pack

Multiple parameters packed into one

```
template<typename ... Args>
void foo(Args ... args)
{
}
```

VARIADIC ARGUMENTS

Parameter Pack Unpacking

Compiler expands the “parameter pack” into a full parameter list

```
template<typename ... Args>
void foo(Args ... args)
{
    SomeObject(args ...);
}
```

VARIADIC ARGUMENTS

Parameter Pack Unpacking

Commonly used in deferred construction

```
template<typename T, typename ... Args>
void std::vector::emplace_back(Args&& args)
{
    data[back_idx++] = T(args ...);
}
```

But can be used in many other advanced situations

Fold Expressions

FOLD EXPRESSIONS

The Problem

So, we know variadic arguments, but how do we do this then?

```
// We have the function signature but...
template<typename ... Args>
void subscribe(void* sub, Args ... args)
{
    // What do we put here?
}
// We still want to achieve this:
subscribe(this, MSG_1, MSG_2, MSG_3, MSG_4);
```

Wouldn’t it be great if we can apply `subscribe()` to each parameter individually?

FOLD EXPRESSIONS

Definition

Allows us to apply the same function to every element in the parameter pack

```cpp
// Assume we already have this:
void bar(int i);

// We can apply this function to all elements like this:
template<typename ... Args>
void foo(Args ... args)
{
    (bar(args), ...);
}

// Now this works:
foo(1, 2, 3, 4);
```

FOLD EXPRESSIONS

Definition

Allows us to apply the same function to every element in the parameter pack

```
// Assume we already have this:
void bar(int i);

// We can apply this function to all elements like this:
template<typename ... Args>
void foo(Args ... args)
{
    (bar(args), ...);
}

// Now this works:
foo(1, 2, 3, 4);
```

Acts as a placeholder for each element in the parameter pack

FOLD EXPRESSIONS

Breakdown

Fold expressions expand into a function call per parameter

```
// Assume we already have this:
void bar(int i);

// We can apply this function to all elements like this:
template<typename ... Args>
void foo(Args ... args)
{
    (bar(args), ...);   →   bar(1), bar(2), bar(3), bar(4);
}

// Now this works:
foo(1, 2, 3, 4);
```

FOLD EXPRESSIONS

The Solution

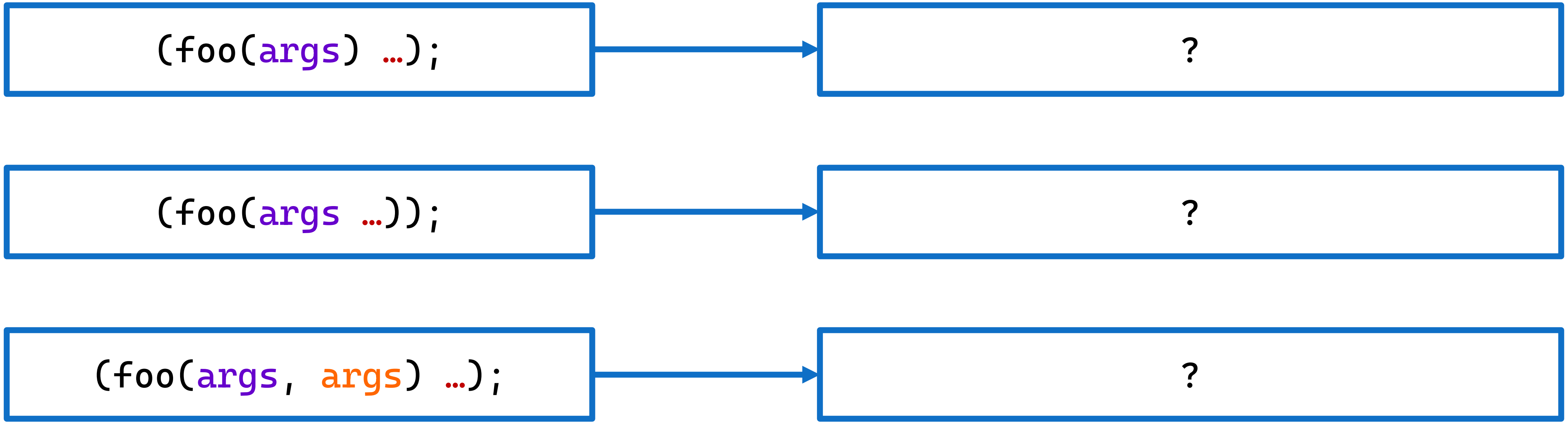
Putting everything together, we get

```
// Create a single parameter version:
void subscribe(void* sub, T msg);

// Update the template version to use it
template<typename ... Args>
void subscribe(void* sub, Args ... args)
{
   (subscribe(this, args), ...);
}

// This now works
subscribe(this, MSG_1, MSG_2, MSG_3, MSG_4);
```

FOLD EXPRESSIONS

Precaution

The location of the expansion operator (…) matters!



```
(foo(args) …);
```

?

```
(foo(args …));
```

?

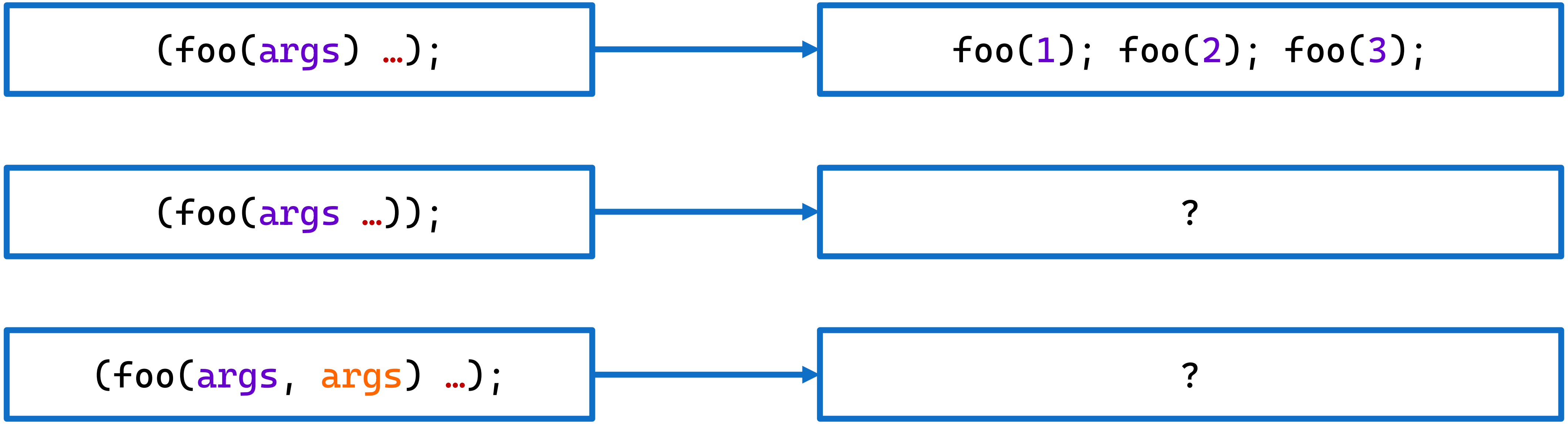
```
(foo(args, args) …);
```

?

FOLD EXPRESSIONS

Precaution

The location of the expansion operator (…) matters!



```
(foo(args) …);
```

```
foo(1); foo(2); foo(3);
```

```
(foo(args …));
```

?

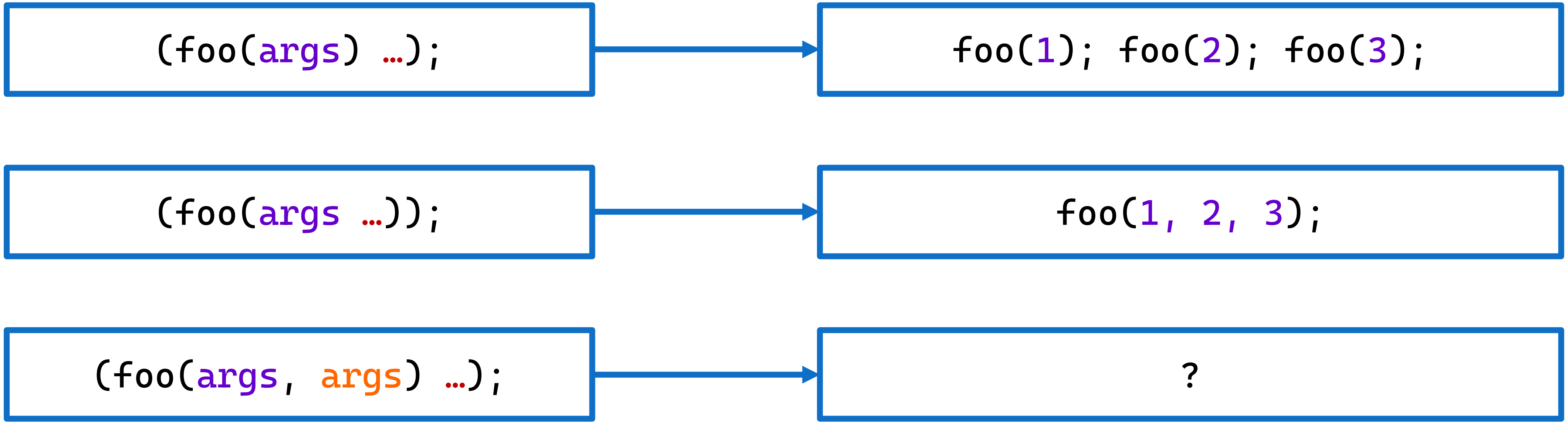
```
(foo(args, args) …);
```

?

FOLD EXPRESSIONS

Precaution

The location of the expansion operator (…) matters!



```
(foo(args) …);
```

```
foo(1); foo(2); foo(3);
```

```
(foo(args …));
```

```
foo(1, 2, 3);
```

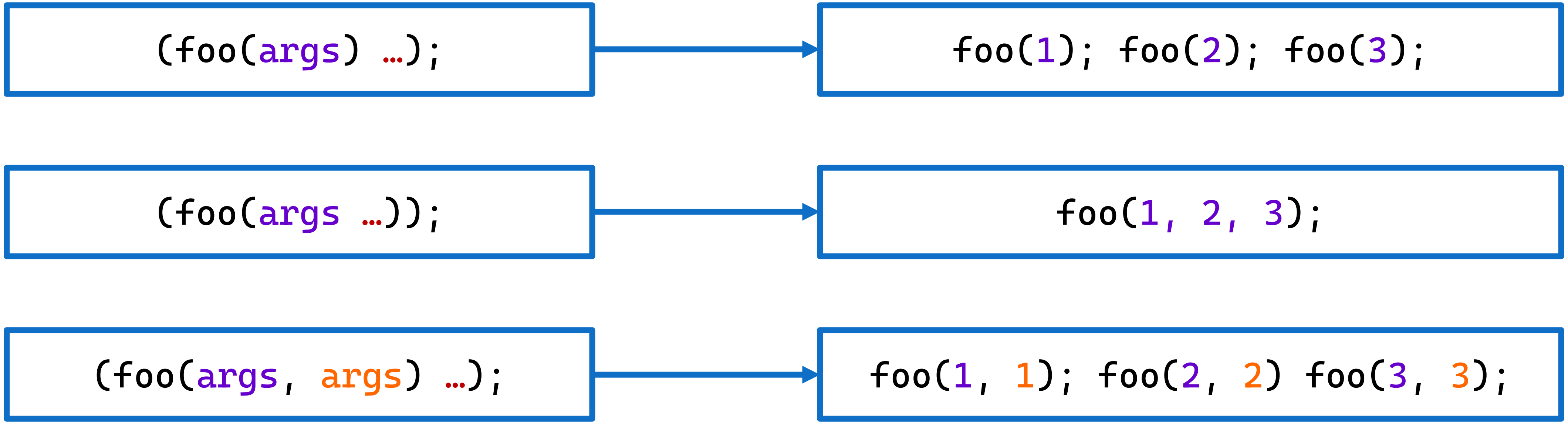
```
(foo(args, args) …);
```

```
?
```

FOLD EXPRESSIONS

Precaution

The location of the expansion operator (…) matters!



```
(foo(args) …);
foo(1); foo(2); foo(3);

(foo(args …));
foo(1, 2, 3);

(foo(args, args) …);
foo(1, 1); foo(2, 2) foo(3, 3);
```

FOLD EXPRESSIONS

With Operators

Works with operators too!

```
const float SUM = args + ...;

const float PRODUCT_SUM = args * ...;
```

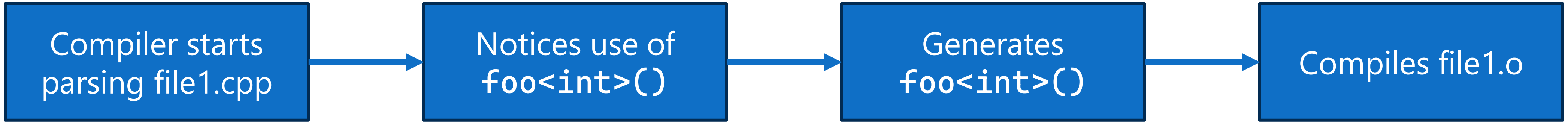
For more details look at parameter packs & fold expressions on cppreference.

Explicit Instantiation

EXPLICIT INSTANTIATION

Recap

How does template instantiation work?



Compiler starts parsing file1.cpp

Notices use of `foo<int>()`

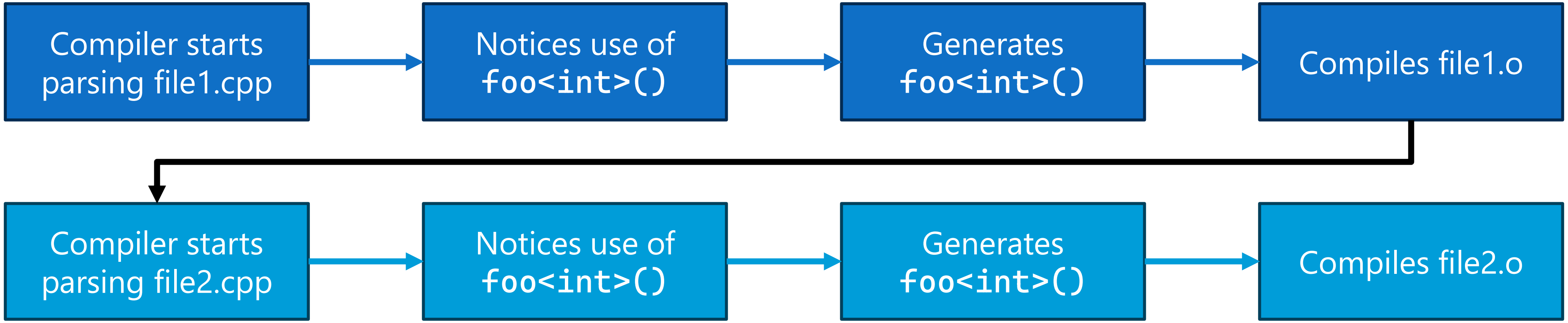
Generates `foo<int>()`

Compiles file1.o

EXPLICIT INSTANTIATION

Recap

How does template instantiation work?



Compiler starts parsing file1.cpp

Notices use of `foo<int>()`

Generates `foo<int>()`

Compiles file1.o

Compiler starts parsing file2.cpp

Notices use of `foo<int>()`

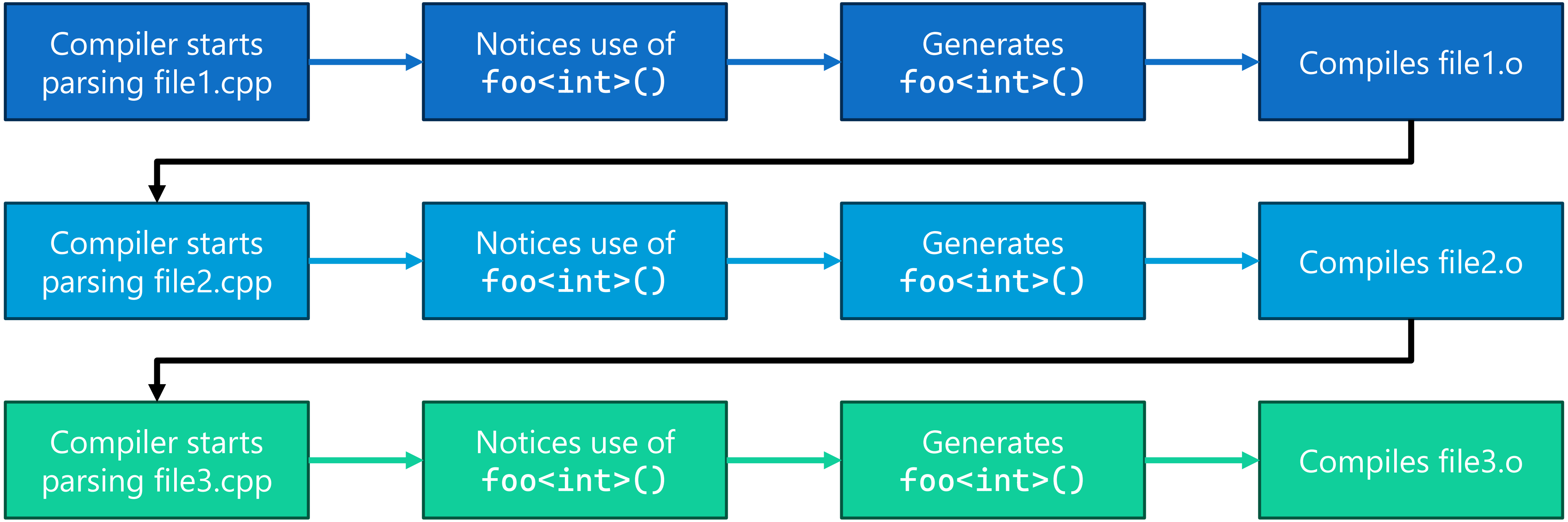
Generates `foo<int>()`

Compiles file2.o

EXPLICIT INSTANTIATION

Recap

How does template instantiation work?



Compiler starts parsing file1.cpp

Notices use of `foo<int>()`

Generates `foo<int>()`

Compiles file1.o

Compiler starts parsing file2.cpp

Notices use of `foo<int>()`

Generates `foo<int>()`

Compiles file2.o

Compiler starts parsing file3.cpp

Notices use of `foo<int>()`

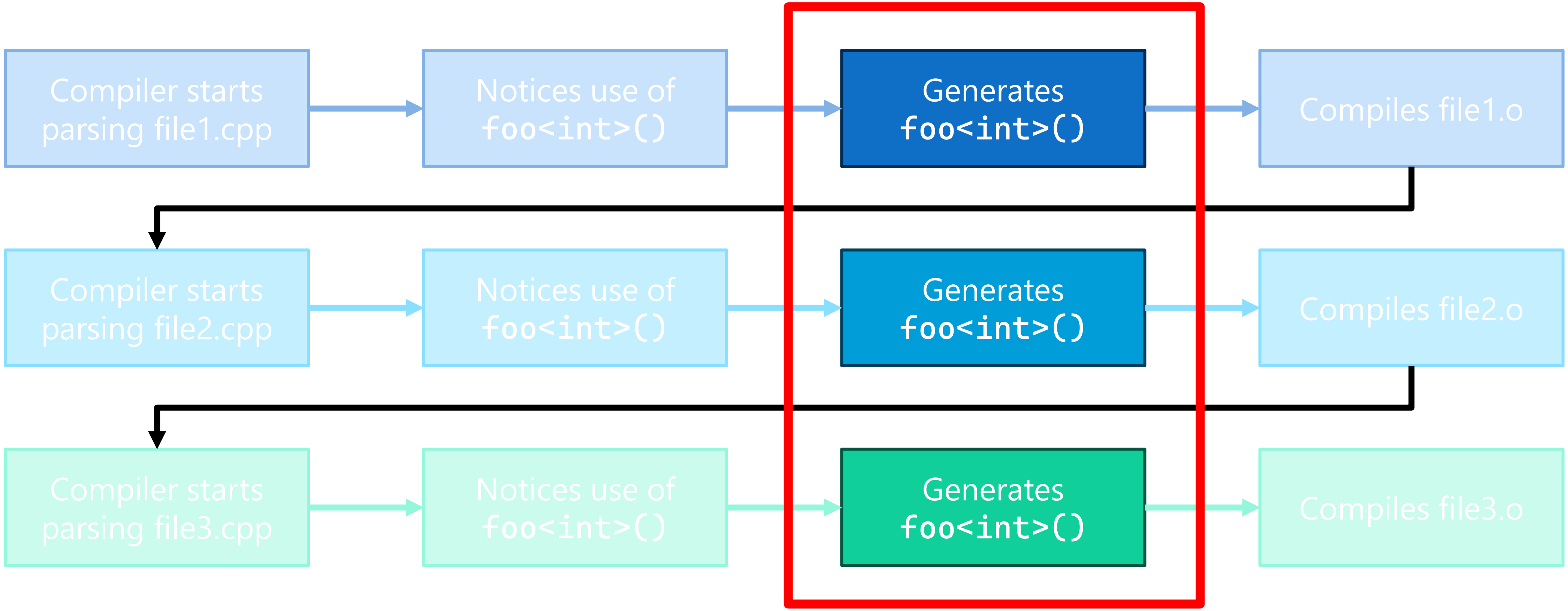
Generates `foo<int>()`

Compiles file3.o

EXPLICIT INSTANTIATION

The Mild Annoyance

Why are we wasting time and space on `foo<int>()` over and over?



Compiler starts parsing file1.cpp

Notices use of `foo<int>()`

Generates `foo<int>()`

Compiles file1.o

Compiler starts parsing file2.cpp

Notices use of `foo<int>()`

Generates `foo<int>()`

Compiles file2.o

Compiler starts parsing file3.cpp

Notices use of `foo<int>()`

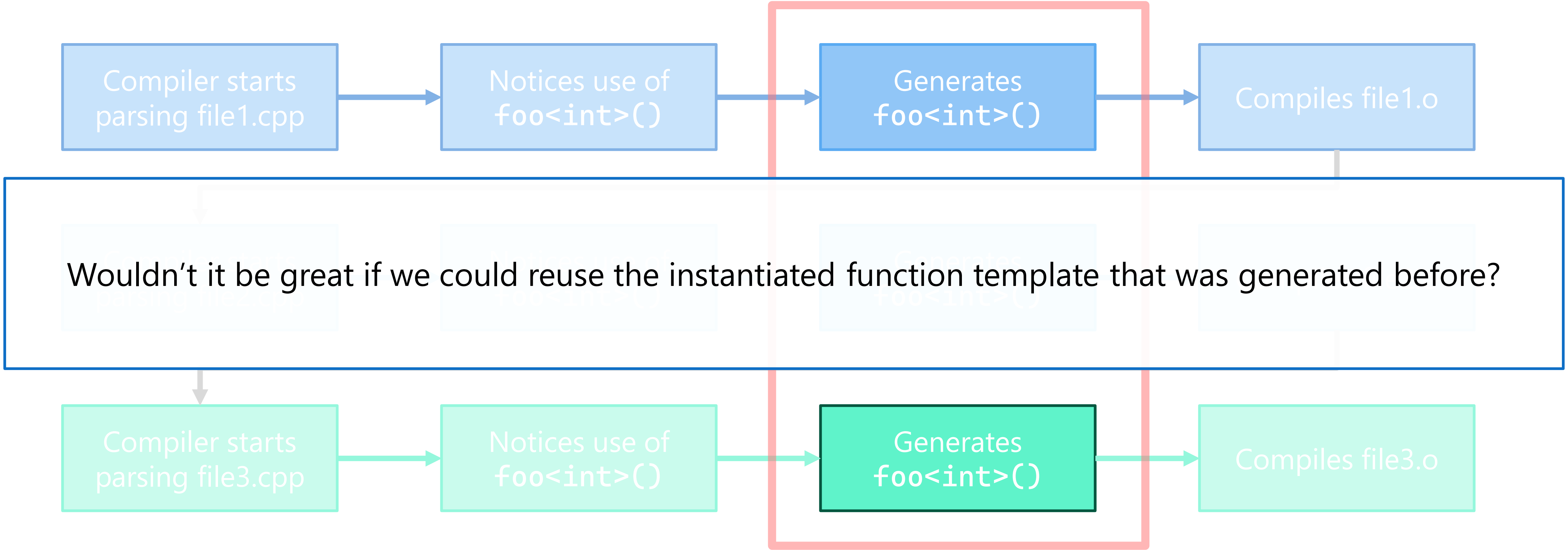
Generates `foo<int>()`

Compiles file3.o

EXPLICIT INSTANTIATION

The Mild Annoyance

Why are we wasting time and space on `foo<int>()` over and over?

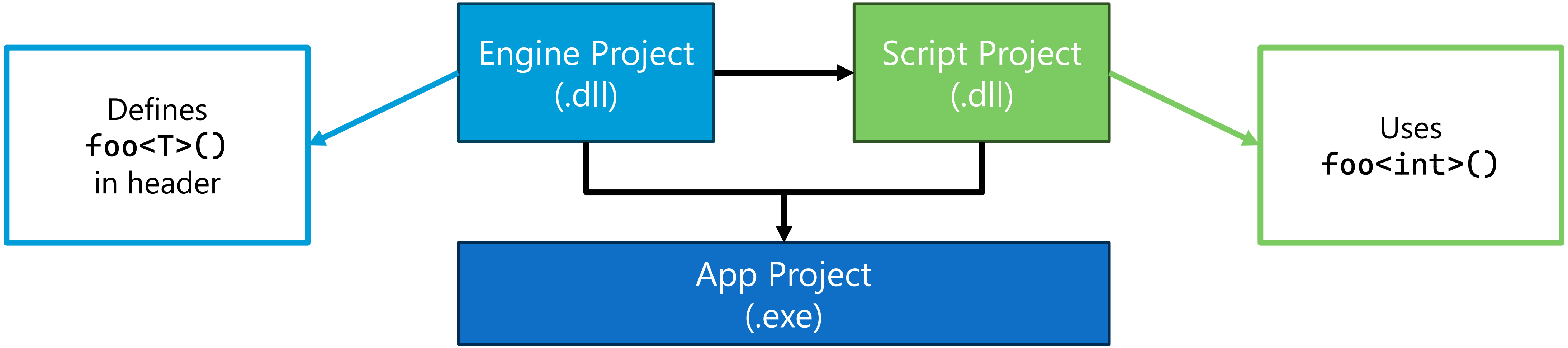

Compiler starts parsing file1.cpp
Notices use of foo<int>()
Generates foo<int>()
Compiles file1.o
Compiler starts parsing file3.cpp
Notices use of foo<int>()
Generates foo<int>()
Compiles file3.o

Wouldn’t it be great if we could reuse the instantiated function template that was generated before?

EXPLICIT INSTANTIATION

The Big Problem

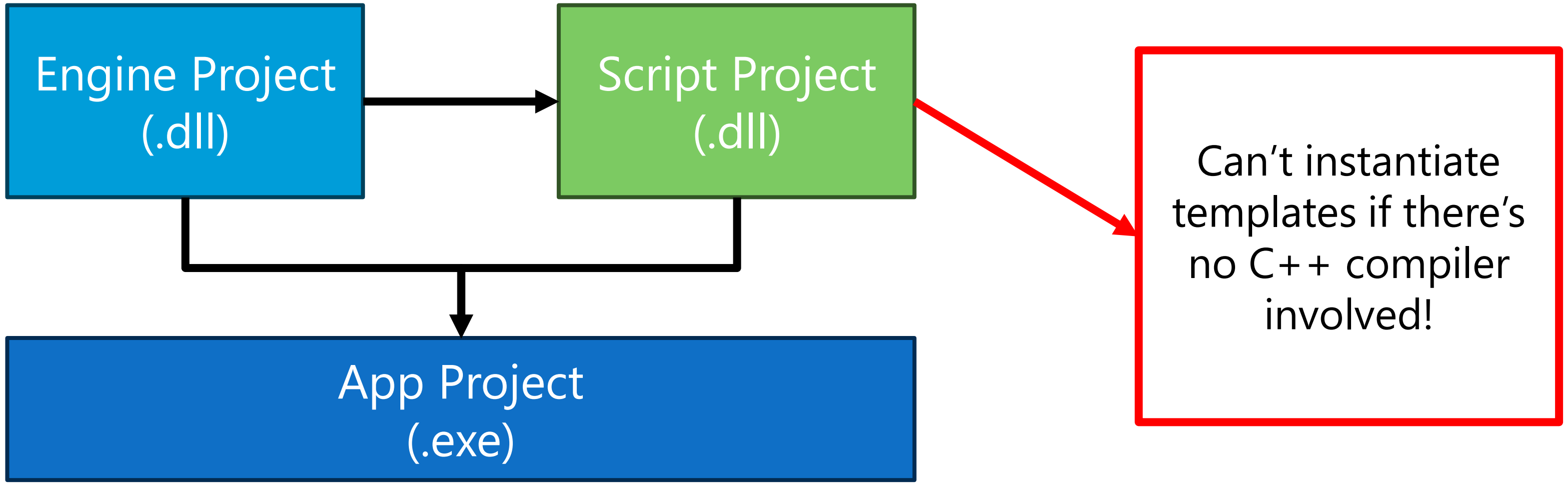
This is the main issue we are more concerned about here



EXPLICIT INSTANTIATION

The Big Problem

Now this causes linker errors!



Can’t instantiate templates if there’s no C++ compiler involved!

EXPLICIT INSTANTIATION

Function Templates

Allows us to instantiate a template ahead of time and insert into the library file

```
// Function template that we need
template<typename T>
T* getComponent<T>(int id)
{
    // Do get component kind of stuff here
}

// Explicit instantiation (in .cpp):
template Renderer* getComponent<Renderer> (int);
```

For more details look at Function Template Instantiation at Microsoft Learn

EXPLICIT INSTANTIATION

Class Templates

Allows us to instantiate a template ahead of time and insert into the library file

```
// Class template that we need
template<typename T>
class Foo
{
    // Class members here
}

// Explicit instantiation (in .cpp):
template class Foo<int>;
```

For more details look at Explicit Instantiation at Microsoft Learn

Thanks for Listening

Any Questions?