

CRAFTING EMBEDDED DOMAIN-SPECIFIC LANGUAGE (EDSL) IN C++ USING METAPROGRAMMING, OPERATOR OVERLOADING, LAMBDA AND MACRO

CONCEPTS, TECHNIQUES, IDEAS, AND APPLICATION EXPLAINED

GILANG MENTARI HAMIDY

<https://heliosky.com> | <http://gilang.hamidy.net>



HI! MY NAME IS **GILANG** MENTARI HAMIDY



E-mail: gilang.hamidy@gmail.com
Web: <https://heliosky.com>
Web2: <http://gilang.hamidy.net>
Twitter/Insta: @GilangHamidy
Telegram: @gmhmd



Bio

- From **Indonesia**, currently living in Finland studying Masters in Security and Cloud Computing at **Aalto University**
- Previously working as a tech consultant (~4 years) at **Accenture**, then move to **Samsung** as Software Engineer (3 years), where I gained extensive experiences in C++
- Interests: Programming Languages, System Software, IoT

WHAT IF...

What if we make asynchronous operation looks better in codes?

- What if we use lambda expression instead of creating callback function?
- Can we form a new kind of statement block to designate an asynchronous operation?

Something like this...

```
void PostListController::LoadData()
{
    // Load post asynchronously
    tfc_async
    {
        // Perform loading from internet
    }
    tfc_async_complete
    {
        // Do something after it's loaded
    };
}
```

WHAT YOU WILL LEARN?

- **Designing EDSL using C++ language construct**
- **Basic concept of EDSL processing**
- **Some metaprogramming recipe for our EDSL language**

PREREQUISITE SKILLS?

- **Basically this topic is for everyone interested in building libraries and frameworks**
 - At least know the fundamental features of C++
- **“Do I need knowledge about compiler construction?”**
Not Really 😊

Basically this topic is for everyone interested in building libraries and frameworks

- Familiar with C++ features including: operator overloading and templates
- Willing to learn template metaprogramming, if never touch that 😊
- Template metaprogramming is not a rocket science, really. I began learning it like only 2 years ago 😊

“Do I need knowledge about compiler construction?”

- Not really, but it is good if you know about it
- I took a class about Compiler Techniques back then, 8 years ago, I got only B-, and didn't really get any knowledge from that class 😊
- This talk will cover a simplified theory about compiler and how we will exploit it in building our EDSL

WHAT IS EDSL

Domain-Specific Language (DSL) is a language designed for a specific application domain or purpose

- More expressive
- Potential gain in productivity
- Reducing maintenance cost
- **Example popular DSL:** HTML, Logo, SQL, etc

WHAT IS EDSL

- **Embedded Domain-Specific Language (EDSL)** is a DSL which is built upon existing programming language infrastructure (*Hudak, 1996*)
- **EDSL:** integrating DSL into your GPL source code

WHY EDSL?

- **Example in Boost Libraries:**
 - **Boost.Spirit:** Creating a parser using sets of grammar directly in C++ codes
- **Syntactic sugar for our custom API or libraries**
- **But EDSL is **not** for everything**
- **Always consider a good design first**

Examples of EDSL:

- There are other examples of EDSL outside of C++, in C#, LINQ can be considered as EDSL, although LINQ is actually an integral part of the language and the compiler

EDSL can introduce a new syntactic sugar within C++ for our custom API or libraries

- Syntactic sugar (with a good documentation, of course) can give a better readable codes compared to boilerplates of C++ codes
- No need to create a separate compiler and separate source files, everything will be dealt with C++ compiler in the same single source files

EDSL is not for everyone:

- Building EDSL can takes longer time than building a normal set of API
- Other way to implement a good and readable API design is to use Fluent API design
 - Fluent Design is simpler to design
 - Does not necessarily require advanced feature of C++

EDSL WE CAN BUILD WITH C++

- **Static/Compile Time EDSL**
 - Covers **boilerplate declarations** and **type definitions**
 - A bit **easier** to build
- **Runtime EDSL**
 - Covers **boilerplate statements** and **function definitions**
 - A bit **harder** to build

Static EDSL:

- Generates compile time products, such as types, generated function, or other compile time objects
- Usually to generate a metadata within the program which will be utilized by standard codes
- Usually implemented simply using macro

Runtime EDSL:

- Executes at runtime and integrate with actual program flow and runtime objects
- To actually perform actions, computations, or processes at runtime
- Implemented by combining operator overloading, function calls, macros etc

**LET'S GET GOING FOR THE HARDER ONE,
SHALL WE? 😊**

CASE: ASYNCHRONOUS CODE FOR TIZEN

- **Tizen:** Uses Enlightenment Framework (EFL) Flat-C API to build apps
- **Drawbacks:** **Procedural languages!**
- **Case:** Application requires async ops for UI responsiveness
 - Modern C++ available, but not aligned with Tizen C API

Tizen: Uses Enlightenment Framework (EFL) Flat-C API to build apps

Drawbacks: Procedural languages has steep learning curves for engineers moving from object-oriented language like Java

Case: Many aspects in application requires asynchronous operation in order to maintain UI responsiveness

- Tizen enables developers to utilize modern C++ (e.g. C++11), and `std::thread` is actually available
- But EFL enforces developers to utilize its own threading API in order for smooth interfacing with EFL UI processing (e.g. main loop, message passing, etc)
- Therefore, `std::thread` infrastructure is not working well

CASE: ASYNCHRONOUS CODE FOR TIZEN

EFL API for Thread

- `ecore_thread_run(Cb* func_thread, Cb* func_end, Cb* func_cancel, void* data);`
 - **Cb***: Function pointer type for callback
 - **void* data**: User data to pass to the callback function
- **Flat C API**
- **No Type Safety (void* data pointer)**

EFL API for Thread

- `ecore_thread_run(Cb* func_thread, Cb* func_end, Cb* func_cancel, void* data);`
 - **Cb***: Function pointer type for callback
 - **void* data**: User data to pass to the callback function
- As the API is Flat-C, we have to pass any state or context through **void* data** parameter, and the callback itself is a regular stateless function
- Imagine an application requiring to connect to internet heavily to obtain data, to maintain responsiveness of the app will **require tons of callback functions and calls to ecore_thread_run**

BASIC DESIGN OF EDSL

- **Creating EDSL can be thought as adding a new “syntax” to an existing programming language**
- **It is good to follow the original style of the programming language to makes the EDSL looks natural**

Creating EDSL can be thought as adding a new “syntax” to an existing programming language

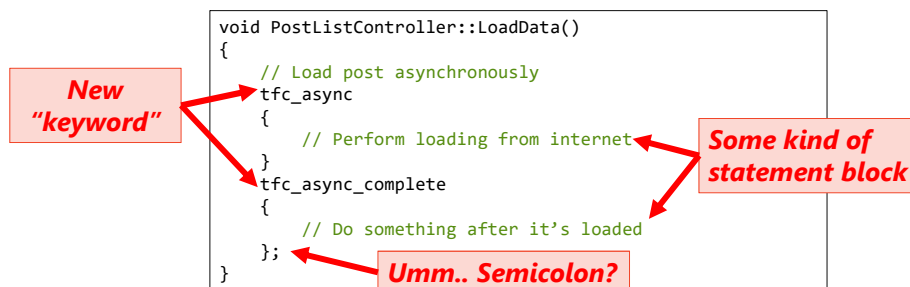
- The syntax can be for any purpose that is not covered by the vanilla programming language
- Feel free to adapt other programming language features or invent something new

Although it is not necessary, it is good to follow the original style of the programming language to makes the EDSL looks natural

- Avoid unnatural uses of operators, parentheses, and semicolon
- Preprocessor Macro can be a good way to “hide” obscure operators

BASIC DESIGN OF EDSL

“New” kind of asynchronous block in C++ code:



We can loosely define the grammar as:

```
tfc_async { statements } tfc_async_complete { statements };
```

Crafting EDSL in C++ | CppCon 2018
Gilang Hamidy | <https://heliosky.com> | @GilangHamidy

14

HOW WE IMPLEMENT THE DESIGN?

```
tfc_async { statements } tfc_async_complete { statements };
```

Using macro?

Using lambda?

- But lambda has a form like this:

```
[ captures ] ( parameter ) { statements };
```

- Hide it inside the macro!

```
#define tfc_async SOMETHING [ ] ( )
```

What is that??

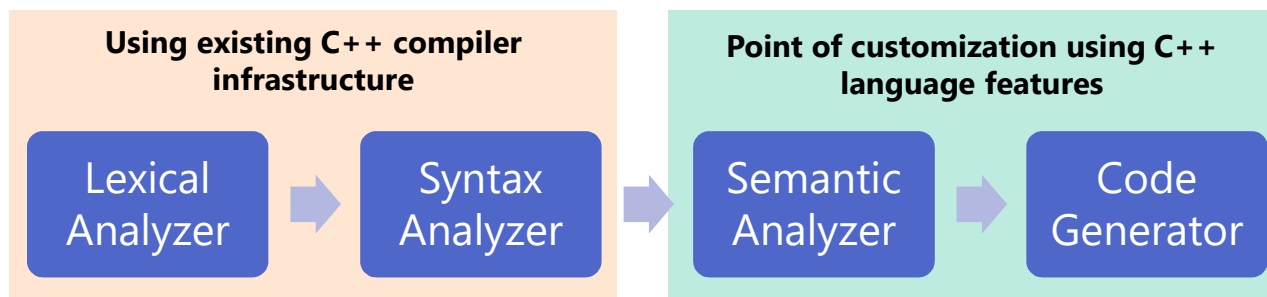
Part of lambda syntax

Crafting EDSL in C++ | CppCon 2018
Gilang Hamidy | <https://heliosky.com> | @GilangHamidy

15

COMPILATION PROCESS

- **Developing EDSL uses similar steps as developing real programming language**
- **Simplified process of Compilers** (*Aho et al, 1988*)



Crafting EDSL in C++ | CppCon 2018
Gilang Hamidy | <https://heliosky.com> | @GilangHamidy

16

Developing EDSL uses similar steps as developing real programming language

- We require a compiler to transform EDSL syntaxes into actual program
- In compiler, source codes passes through several processes before it will be transformed into executables

Steps:

1. **Lexical Analyzer**
 - Perform tokenization of Input Sources
2. **Syntax Analyzer**
 - Parse input token to validate the grammar of the source
 - Build **parse tree** which represents the source for semantic analysis
3. **Semantic Analyzer**
 - Analyze the syntax tree and identify the meaning of the program
 - Validate the meaning of the program
4. **Code Generator**
 - Generate target code based on information produced by Semantic Analyzer

Using existing C++ compiler infrastructure

- We do not need to reimplement lexical and syntax analyzer because our EDSL will be integrated within C++ language itself
- The EDSL will utilize existing C++ tokens and syntaxes with minimal syntactic modification via preprocessor (macros)

Point of customization using C++ language features

- C++ enables us to modify the semantic of a program via its inherent feature: **operator overloading**
- We can perform validation and generate target actions based on our own custom defined semantic

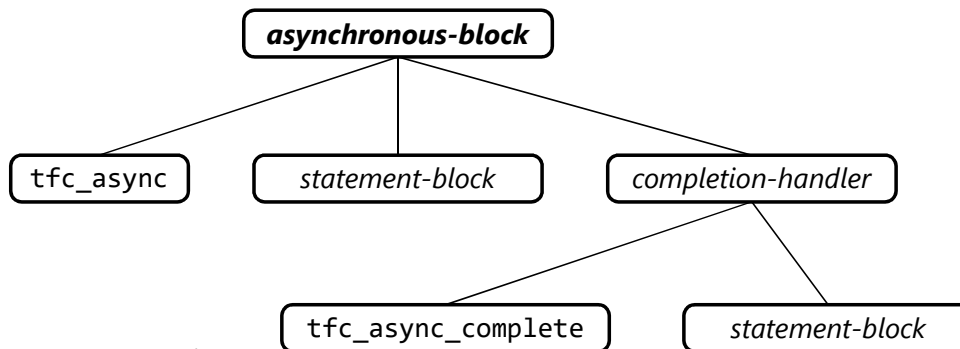
EXAMPLE OF PARSE TREE: HYPOTHETICAL TREE FOR TFC_ASYNC

Formal Grammar:

asynchrononous-block → *tfc_async* *statement-block* *completion-handler*

statement-block → { *statements-list* }

completion-handler → *tfc_async_complete* *statement-block*



**We need to build
this parse tree to
process our actual
EDSL**

BUILDING PARSE TREE USING OPERATOR OVERLOADING

- **Operator Overloading enables us to build and analyse parse tree of our EDSL**
- **Operator can be used in any expression context, and not tight to specific form such as assignment, function call, etc. Therefore, this expression is valid in C++:**

```
someObjectA + someObjectB;
```

- **Operator Overload:**

As Static Function

```
ResultType operator& (Operand1 o1, Operand2 o2);
```

As Member Function

```
class Operand1  
{  
    ResultType operator& (Operand2 o2);  
};
```

Crafting EDSL in C++ | CppCon 2018
Gilang Hamidy | <https://heliosky.com> | @GilangHamidy

18

Operator Overloading enables us to build and analyse parse tree of our EDSL

- Operator overloading provides access to modify the semantics of operators and expressions
- We can customize the behaviour of an operator to follow our custom rules for defining our EDSL
- Our EDSL must be designed to conform the C++ operator and expression rules

Operator can be used in any expression context, and not tight to specific form such as assignment, function call, etc.

Properties of Operator Overloading for processing EDSL

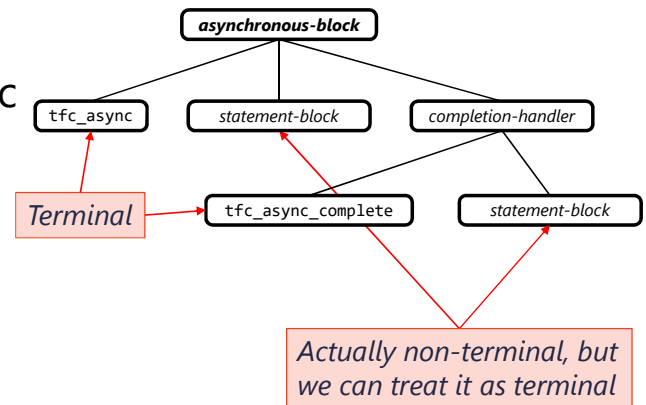
- Operator precedence is used to direct how our EDSL is processed
- Expression operands are the data supplied by the programmer to the EDSL processor

How do we overload operator?

- Either as static function or as member function
- Depends on the context, we can pick both. But the important part is that the EDSL terminal will be represented as **operator operands**

TERMINAL SYMBOL

- **Terminal: symbol that may appears in a language**
 - Keywords, identifiers, literals, etc
 - terminal is represented as **leaf**
- **Need terminal symbol for our EDSL**
 - **Cannot reuse existing keywords**



Crafting EDSL in C++ | CppCon 2018
Gilang Hamidy | <https://heliosky.com> | @GilangHamidy

19

Terminal is a literal symbol that may appears in a language

- Terminal can be keywords, identifiers, literals, etc
- In Parse Tree, terminal is represented as **leaf**
- In building EDSL, we have to create additional symbol which will indicates our EDSL syntax

We need a terminal symbol for our EDSL

- We **cannot** reuse existing keywords for our terminal symbol
- All other word appear in C++ codes (apart of literals) will be dealt as identifier, and must represent either a class name, function name, or variable name

Therefore, we create User Defined Types (class/struct) to represent our terminal symbols

- The type itself will act **as a terminal** for our parse tree
- The new type does not necessarily has any purpose other than a placeholder symbol for a terminal in a parse tree
- C++ compiler will see the type as just another identifier

Then, we need to pick which operator we will overload

- We need to pick the operator based on **operator precedence**, which therefore will force the compiler to build correct parse tree for us
- Reference of operator precedence:
https://en.cppreference.com/w/cpp/language/operator_precedence
- **Note that** not all operators are overloadable

TRICKS USING LAMBDA FOR COMPOUND STATEMENTS

- **In C++, compound statement can appears anywhere inside a function block**
 - **But insensitive with the context**
- **Lambda Statement provides a mechanism to introduce a compound statement as a callable object**

In C++, compound statement can appears anywhere inside a function block

- But compound statement usually insensitive with the context unless it is defined in the grammar, such as: conditional statement, loop statement, etc
- Unfortunately we cannot change the grammar of C++ unless we change the compiler (and the standards)

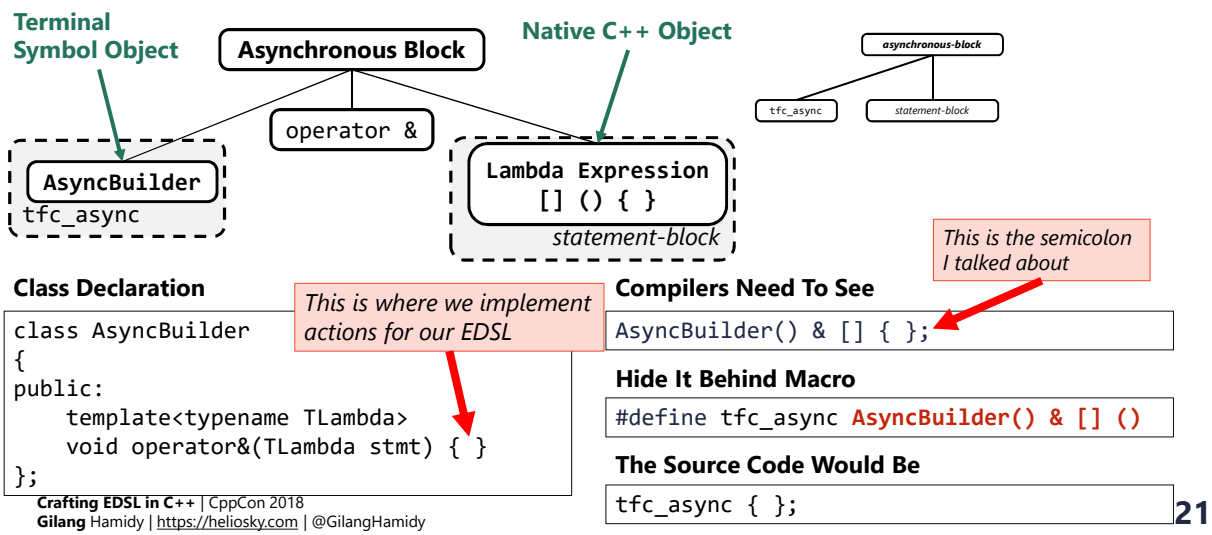
Lambda Statement provides a mechanism to introduce a compound statement as a callable object

- Statements inside a lambda block will behaves exactly like a function block, and it creates its own context when it is called
- It can mimics the looks of a compound statement, while allowing us to incorporate it in our semantic processing

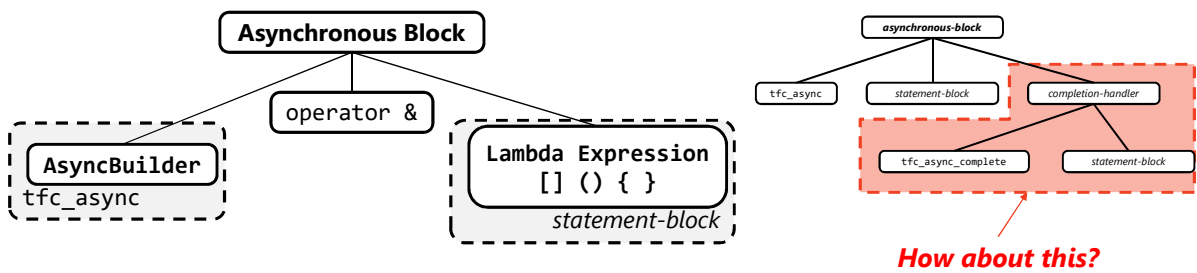
Then we can make statement-block considered as terminal in our EDSL

- As C++ compiler sees Lambda as production rule which eventually becomes an *r-value*, we can directly use that value as input for our operator overloading operand.
- We does not have to take care how C++ will process the compound statement, and we only care for the lambda object and what we are going to do with it

EXAMPLE OF PARSE TREE:
ACTUAL PARSE TREE FOR IMPLEMENTATION



EXAMPLE OF PARSE TREE:
ACTUAL PARSE TREE FOR IMPLEMENTATION



NON-TERMINAL PRODUCTION

- **If a grammar rule does not entirely consist of terminal, it is considered as non-terminal**
- **Overloaded operator which is used for non-terminal production has to produce a temporary object as its return value**

If a grammar rule does not entirely consist of terminal, it is considered as non-terminal

- In parse tree, non-terminal production is represented as a non-leaf node

Overloaded operator which is used for non-terminal production has to produce a temporary object as its return value

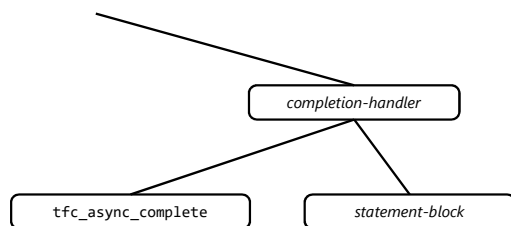
- The temporary object contains all necessary information from its operands to be carried upwards to higher grammar rule
- It also carries the semantic of the EDSL statement
- The temporary object is then propagated to lower-precedence operator overload

Note that in C++, overloadable operator has only one or two operands

- Therefore, we need to simplify production rules to only expand into two children

IMPLEMENTING PARSE TREE OPERATOR OVERLOADING FROM BOTTOM TO TOP

**Begin by building the bottom-most grammar rules, and
return a temporary object for its production**



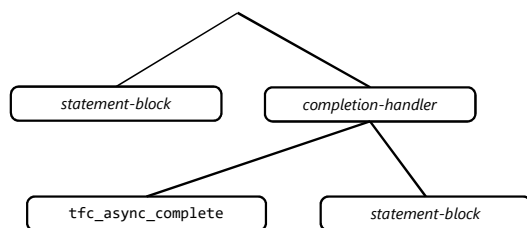
```
class AsyncCompleteBuilder
{
public:
    template<typename TLambda>
    CompleteOperand operator*(TLambda stmt)
    {
        ...
    }
};
```

**To correctly identify each temporary objects and operators to be used, we should to
implement the parse tree from bottom to top**

**We begin by identifying the deepest level of terminals in our parse tree, then
implement the operator overload according the grammar's requirement**

IMPLEMENTING PARSE TREE OPERATOR OVERLOADING FROM BOTTOM TO TOP

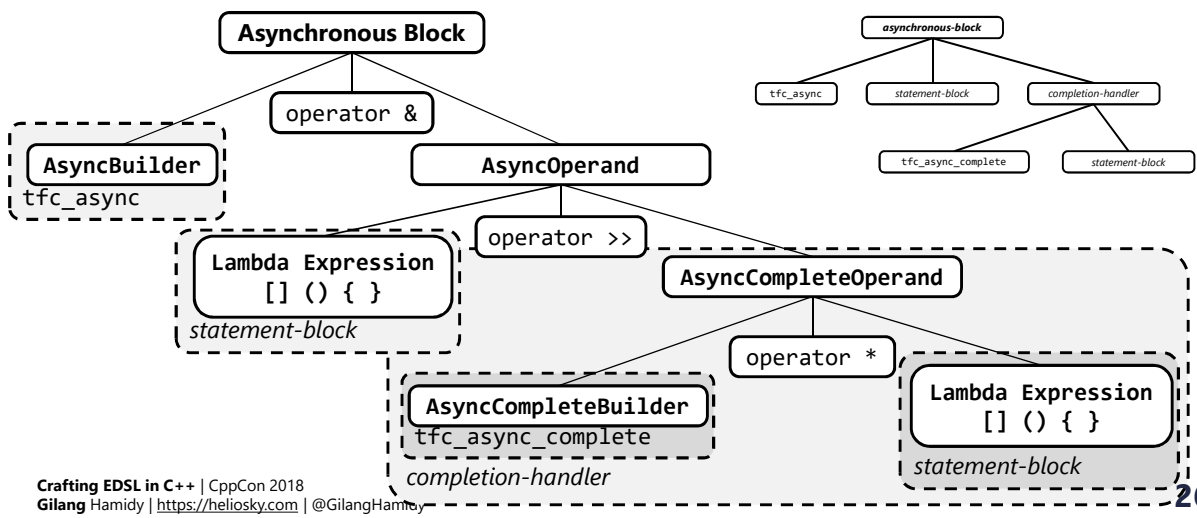
Use the return value of the operator overload, as a lower-precedence operator's operand



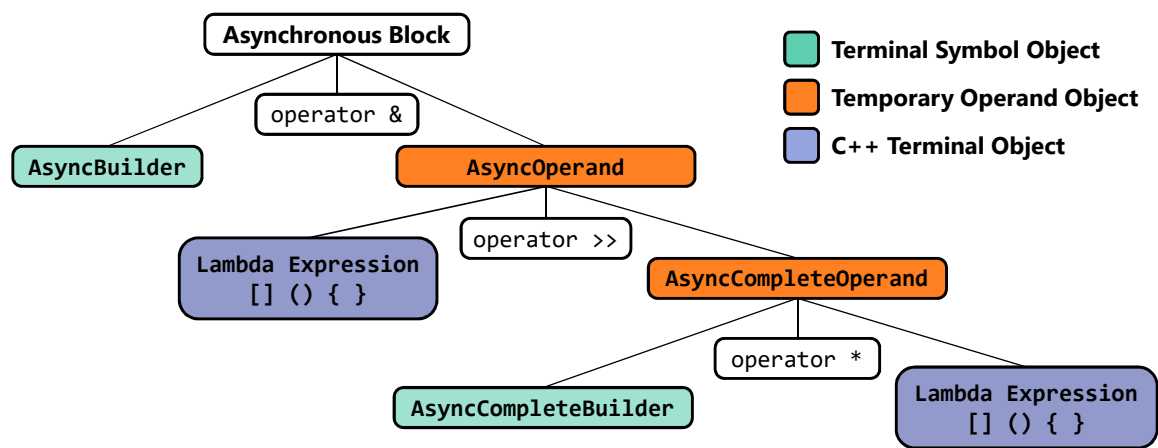
```
template<typename TLambda>
AsyncOperand operator>>(
    TLambda lambda,
    CompleteOperand cplt)
{
    ...
}
```

As operator overload can only has two operands, therefore we need tie the CompleteOperand object to statement-block terminal

EXAMPLE OF PARSE TREE:
ACTUAL PARSE TREE FOR IMPLEMENTATION



EXAMPLE OF PARSE TREE:
ACTUAL PARSE TREE FOR IMPLEMENTATION



CARRYING INFORMATION UPWARDS

- **Information from lower level of parse tree needs to be carried upwards and processed by higher level nodes**
- **In our case, we need to carry our lambda upwards to the root of the parse tree. There are several possible ways:**
 - Using `std::function`
 - Using template

Information from lower level of parse tree needs to be carried upwards and processed by higher level nodes

- The processing may happen in any operator overloads, or may be transferred upwards towards the root of the parse tree
- The information itself is stored inside the Temporary Operand Object and passed as return value of the overloaded operator

In our case, we need to carry our lambda upwards to the root of the parse tree. There are several possible ways:

- Using `std::function`
- Using template

CARRYING LAMBDA WITHIN TEMPORARY OBJECT USING TEMPLATE

**Using template can retain type information in the temporary object,
and we can use template metaprogramming to analyze the type
further (if needed)**

```
template<typename TLambda, typename TCompleteLambda>
struct AsyncOperand
{
    TLambda lambda;
    TCompleteLambda completeLambda;
};

template<typename TCompleteLambda>
struct CompleteOperand
{
    TCompleteLambda lambda;
};
```

Gilang Hamidy | <https://heliosky.com> | @GilangHamidy

29

WHAT THE COMPILER NEEDS TO READ

- **Finally, compiler needs to read this to process our EDSL:**

```
AsyncBuilder() & [] () { } >> CompleteBuilder() * [] () { };
```

- **With proper preprocessor:**

```
#define tfc_async AsyncBuilder() & [] ()
#define tfc_async_complete >> CompleteBuilder() * []
```

- **We can have this syntactic sugar:**

```
tfc_async { } tfc_async_complete { };
```

Crafting EDSL in C++ | CppCon 2018
Gilang Hamidy | <https://heliosky.com> | @GilangHamidy

30

VALIDATING EDSL

- **We can use template metaprogramming to direct correct EDSL semantics**
 - Using `type_traits` or other metaprogramming rules, and `static_assert` to fail compilation
- **Let's enhance our lambda by implementing rules!**
 - How if we use return statement from lambda, to be passed into the second lambda which will be called after asynchronous lambda has been completed

Crafting EDSL in C++ | CppCon 2018
Gilang Hamidy | <https://heliosky.com> | @GilangHamidy

31

We can use template metaprogramming to direct correct lambda expression to be supplied by users

- Lambda has to be validated before the EDSL can accept it as a correct semantics
- We can detect the correctness during compilation time using template metaprogramming, including type traits
- If the validation fails, the compiler can raise compile error, helped by `static_assert`

OUR LAMBDA RULES


- **Example of lambda rules for `tfc_async`:**
 1. Async handler: May return value
 2. Completion handler: 0 or 1 parameter
 3. Async return type == Completion parameter type

- Example of lambda rules for `tfc_async`:
 1. First lambda, the asynchronous handler may return a value
 2. Second lambda, the completion handler, may have none or single parameter, and does not return a value
 3. Return value of the asynchronous lambda must be the same with parameter of the completion lambda


WHAT THE ENHANCED LAMBDA WILL LOOKS LIKE

```
void PostListController::LoadData()
{
    // Load post asynchronously
    tfc_async
    {
        // Perform loading from internet
        std::string someData = getFromInternet();
        return someData;
    }
    tfc_async_complete (std::string theData)
    {
        // Do something after it's loaded
    };
}
```

***"return" the
data to the
main thread***



***Capture the data
from the
asynchronous
thread***



IMPLEMENTING VALIDATION RULES

- **Unfortunately, current C++ library does not provide a type trait to detect a callable objects**
- **Detecting if a type has a call operator**

```
template<typename T>
class HasCallOperator
{
    typedef char Correct;
    typedef long Incorrect;
    template<typename TTest> static Correct Test(decltype(&TTest::operator()));
    template<typename TTest> static Incorrect Test(...);
public:
    static constexpr bool Value = sizeof(Test<T>{0}) == sizeof(Correct);
};
```

Crafting EDSL in C++ | CppCon 2018
Gilang Hamidy | <https://heliosky.com> | @GilangHamidy

34

Boost's Type Traits also does not provide this functionality (to my knowing), therefore, let's create one!

Using SFINAE rules

IMPLEMENTING VALIDATION RULES

Detect pointer-to-member-function args and return type

Base template case for all other types

```
template<typename TFuncType>
struct MemberFunction { };
```

IMPLEMENTING VALIDATION RULES

Detect pointer-to-member-function args and return type

Template definition for a pointer-to-member type

```
template<typename TClass, typename TReturn, typename... TArgs>
struct MemberFunction<TReturn (TClass::*)(TArgs...)>
{
    static constexpr auto Arity = sizeof...(TArgs);
    typedef TReturn ReturnType;
    typedef TClass DeclaringType;

    template<size_t idx>
    using Args = typename std::tuple_element<idx, std::tuple<TArgs...>>::type;

    typedef std::tuple<TArgs...> ArgsTuple;
    typedef std::tuple<typename std::decay<TArgs>::type...> ArgsTupleDecay;
};
```

IMPLEMENTING VALIDATION RULES

Detect pointer-to-member-function args and return type

Case for support of const member function

```
template<typename TClass, typename TReturn, typename... TArgs>
struct MemberFunction<TReturn (TClass::*)(TArgs...) const> :
    MemberFunction<TReturn (TClass::*)(TArgs...)> { };
```

IMPLEMENTING VALIDATION RULES

Detect callable objects (i.e. lambda)

Base template case for all other types

```
template<typename T, bool = HasCallOperator<T>::Value>
struct CallableObject;
```

IMPLEMENTING VALIDATION RULES

Detect callable objects (i.e. lambda)

Base template case for all other types

```
template<typename T, bool = HasCallOperator<T>::Value>
struct CallableObject;
```

IMPLEMENTING VALIDATION RULES

Detect callable objects (i.e. lambda)

Specialization for non callable object (bool is false)

```
template<typename T>
struct CallableObject<T, false>
{
    static constexpr bool Callable = false;
};
```

IMPLEMENTING VALIDATION RULES

Detect callable objects (i.e. lambda)

Specialization for callable object (bool is true)

```
template<typename T>
struct CallableObject<T, true> :
    MemberFunction<decltype(&T::operator())>
{
    static constexpr bool Callable = true;
};
```

VALIDATION RULES BOOLEAN

Computing the validation rules using the metaprogram:

Using statement for type traits

```
using IntrospectL = CallableObject<TLambda>;
using IntrospectCL = CallableObject<TCompleteLambda>;
```


VALIDATION RULES BOOLEAN

Computing the validation rules using the metaprogram:

0 or 1 parameter and return void in complete lambda

```
static_assert(IntrospectCL::Arity == 1
              || IntrospectCL::Arity == 0, "...");

static_assert(std::is_same_v<
              typename IntrospectCL::ReturnType,
              void>, "...");
```

VALIDATION RULES BOOLEAN

Computing the validation rules using the metaprogram:

Async return type == completion param

```
static constexpr bool validation =
    (std::is_same_v<typename IntrospectL::ReturnType, void>
     && IntrospectCL::Arity == 0)
    || std::is_same_v<typename IntrospectL::ReturnType,
                      typename IntrospectCL::template Args<0>>;

static_assert(validation, "...");
```

STATIC_ASSERT IN TEMPORARY OBJECTS

We can put `static_assert` in temporary object

```
template<typename TCompleteLambda>
struct CompleteOperand
{
    using Introspect = CallableObject<TCompleteLambda>;

    // Validation rules in the class definition
    static_assert(Introspect::Arity == 1 || Introspect::Arity == 0, "...");
    static_assert(std::is_same_v<typename Introspect::ReturnType, void>, "...");

    TCompleteLambda lambda;
};
```

TESTING THE RULES AT RUNTIME

Example: Returning value from async block but not “capturing” it on complete block

Error Message (clang):

```
tfc_async
{
    long long ret = 123234252;
    return ret;
}
tfc_async_complete
{
};
```

```
Slide39.cpp:86:5: error: static_assert failed due to requirement 'validation' "The asynchronous
return value and complete lambda has different parameter"
    static_assert(validation, "The ...");
    ^
Slide39.cpp:150:5: note: in instantiation of template class 'AsyncOperand<(lambda at
Slide39.cpp:145:5), (lambda at Slide39.cpp:150:5)>' requested here
    tfc_async_complete
    ^
Slide39.cpp:139:28: note: expanded from macro 'tfc_async_complete'
#define tfc_async_complete >> CompleteBuilder() * []
```

LIMITATION OF EDSL WITH OPERATOR OVERLOADING AND MACRO

- **Increases compilation time!**
- **Need to follow C++ syntax rules and limitation,**
- **Difficult to describe EDSL grammar using operator overloading**
- **Macro does not able to carry syntactic and semantic information**

Need to follow C++ syntax rules and limitation, including limitation in template metaprogramming

Sometimes it is difficult to describe EDSL grammar using operator overloading

Macro does not able to carry syntactic and semantic information, only covers up the obscure operator overload boilerplates

Difficulties to describe EDSL grammar using operator overloading:

- Parse tree needs to be modelled as binary tree.
- Complex grammar with complex parse tree structure must be transformed into binary tree.
- Only limited amount of operators are available

Issues with Macro in EDSL

- Errors for anything behind the macro can creates weird error message obscure to end-user

WRAP-UP

- **Operator overloading to simulate parse tree processing**
- **Modern C++ features for our EDSL**
- **Template metaprogramming for validation**
- **Macros to make the code sweet**

- Developing complex runtime EDSL mainly requires operator overloading to simulate parse tree within a programming language
- We can incorporate lambda expression, custom types, and other literal types in our EDSL
- Template metaprogramming is useful to manipulate semantics of an EDSL, as well as performing validation of our EDSL structure
- Macros are useful to cover up obscure-but-required notation within our EDSL grammar to produce a better syntactic sugar for the user

FINALLY...

**You don't have to be in C++ Standard
Committee to add new "syntax" in C++**
(Although it is super cool if you are a part of them :D)

**C++ is a sandbox and be creative in our
projects!**

Crafting EDSL in C++ | CppCon 2018
Gilang Hamidy | <https://heliosky.com> | @GilangHamidy

49

QUESTIONS?

In the meantime, you can also connect with me... ☺

E-mail: [**gilang.hamidy@gmail.com**](mailto:gilang.hamidy@gmail.com)

Web: [**https://heliosky.com**](https://heliosky.com)

Web2: [**http://gilang.hamidy.net**](http://gilang.hamidy.net)

Twitter/Instagram: **@GilangHamidy**

Telegram: **@gmhmd**

Crafting EDSL in C++ | CppCon 2018
Gilang Hamidy | <https://heliosky.com> | @GilangHamidy

50

ACKNOWLEDGEMENTS

The Framework Team and Tizen Team

Kevin Winata, Peradnya Dinata, Juan Fernando,
Alexius Alvin, and the rest of SRIN Tizen Team

Special Thanks

- Risman Adnan
- Yanuar Rahman



Thanks to **Pablo Halpern** for feedback of the slides

Andrei Alexandrescu, **Kate Gregory**, and **Scott Meyer** for the
AMAZING workshop

and anonymous program chairs and reviewers for suggestions and
acceptance to #CppCon 😊

Crafting EDSL in C++ | CppCon 2018
Gilang Hamidy | <https://heliosky.com> | @GilangHamidy

51

TERIMA KASIH

thank you • kiitos • gracias • matur nuwun • danke • merci
ありがとう • 谢谢 • dank je • grazie • tack • شكرا
спасибо • धन्यवाद • matur suksma • 고맙습니다 • obrigado

THEORETICAL REFERENCE

Marjan Mernik, Jan Heering, and Anthony M. Sloane. 2005. *When and how to develop domain-specific languages*. ACM Comput. Surv. 37, 4 (December 2005), 316-344.
DOI=<http://dx.doi.org/10.1145/1118890.1118892>

Alfred V. Aho, Ravi Sethi, Jeffrey D. Ullman. 1988. *Compilers Principles, Techniques, and Tools*. Addison Wesley.

ABSTRACT

Ever thinking of creating your own specific-purpose programming language? Or maybe improving programming experience by adding a language feature in existing language? This talk presents about Embedded Domain-Specific Language (EDSL) from basic concepts and technique to build it using C++.

EDSL is a powerful technique in integrating a custom libraries or frameworks in users' codes. EDSL can improve user's experience in using libraries by reducing boilerplate codes and providing more readable and straightforward codes. Some success stories are: Boost.Spirit, Boost.Proto.

Although Boost.Proto provides tool to build EDSL in C++, this talk presents building EDSL from the ground up utilizing operator overloading, template metaprogramming, and macro. We use operator overloading and template metaprogramming to build a simple expression tree for our EDSL.

This talk provides an example of developing an "asynchronous block" within C++ codes to be used in Tizen native application development. The "asynchronous block" has similar purpose with C++11 `std::async` but targeting a different API and adapting the structure of try-catch block.