Extend clang to substitute template argument type sugar when performing member access on template specialization - GSoC 2022

Matheus Izvekov <mizvekov@gmail.com>

Mentors: Vassil Vassilev <vvasilev@cern.ch>, Richard Smith <richard@metafoo.co.uk>

Introduction

The LLVM Project is a collection of modular and reusable compiler and toolchain technologies. Despite its name, LLVM has little to do with traditional virtual machines. The name "LLVM" itself is not an acronym; it is the full name of the project. The LLVM Core libraries provide a modern source- and target-independent optimizer, along with code generation support for many popular CPUs (as well as some less common ones!). These libraries are built around a well specified code representation known as the LLVM intermediate representation ("LLVM IR"). The LLVM

Core libraries are well documented, and it is particularly easy to invent your own language (or port an existing compiler) to use LLVM as an optimizer and code generator.

Clang is an "LLVM native" C/C++/Objective-C compiler, which aims to deliver amazingly fast compiles, extremely useful error and warning messages and to provide a platform for building great source level tools. The Clang Static Analyzer and clang-tidy are tools that automatically find bugs in your code, and are great examples of the sort of tools that can be built using the Clang frontend as a library to parse C/C++ code.

Clang's expressive error diagnostics systems can be enhanced further. For an example like:

```
template<class T> struct Foo { using type = T; };
using Bar = int;
Foo<Bar>::type baz;
```

In clang, **baz**'s type will not desugar to **Bar**, as the member access in `Foo<Bar>::type` will go directly for the canonical type of the template instantiation. This will cause any diagnostics printed from this type to miss this information. If there were an attribute attached to the **Bar** typedef, such as alignment, this would be lost on the **baz** declaration.

In this proposal, we want to lift this restriction by implementing the missing functionality in clang.

Overview

The basic solution to this problem will be to create a new type node which represents the sugar of a member access in a template specialization. When performing a single step desugar of this new type node, the implementation should substitute any sugar for template parameters into the as-written type used in the corresponding argument of the template specialization.

Goals

1) Implement a type node which will hold the sugar for member access.
2) Implement single step desugaring logic which will perform the substitution of template parameter sugar into the corresponding specialization argument sugar.

Implementation Approach

The ClassTemplateSpecializationDecl represents the canonicalized result of forming the specialization, so

```
using Foo = int;
vector<Foo> x;
vector<int> y;
auto v = x.begin();
```

... will result in x and y denoting the same ClassTemplateSpecializationDecl. It doesn't make sense for that declaration to track the sugared type, because that declaration is used for accesses with different sugar. The type sugar should not ever affect the behavior of the instantiation, so should not be part of the instantiation. Rather, it's part of the way that the instantiation is named, so should be part of that naming context.

For example, when we form the "x.begin" expression in the above example (in BuildMemberReferenceExpr or somewhere near there), we could:

- Take the type T1 of 'x':
  `-TemplateSpecializationType 0xdafa140 'vector<Foo>' sugar vector
    |-TemplateArgument type 'Foo':'int'
    `-RecordType 0xdafa120 'vector<int>'
    `-ClassTemplateSpecialization 0xdafa040 'vector'

- Take the type T2 of 'begin':
  `-FunctionProtoType 0xdb287b0 'vector<int, allocator<int>>::iterator ()' cdecl
    `-RecordType 0xdb28780 'vector<int, allocator<int>>::iterator'
    `-CXXRecord 0xdafa428 'iterator'

- Form a new type sugar node (say, MemberSugarType) that tracks T1 and T2, and has the same canonical type as T2, but that has custom desugaring logic to allow it to produce a more friendly type name

In this case, desguaring a MemberSugarType wrapped around a FunctionProtoType would push the member sugar onto the return type and parameter types of the type; desugaring a MemberSugarType wrapped around a RecordType representing a member of the type tracked by a MemberSugarType would form an ElaboratedType describing the name of that member as a member of the sugared type. So we'd end up with the type of "x.begin" desugaring to vector<Foo>::iterator(), as desired. (When we grab the return type of a function in a function call, we'll need to be careful to single-step desugar it until we reach a function type, to move the sugar onto the return type. And likewise in other places too.)

Likewise, for x.front(), where the type of 'front' is something like:

```
        `-FunctionProtoType 0xdb3a0b0 'vector<int, allocator<int>
> ::value_type ()' cdecl
        `-TypedefType 0xdb3a060 'vector<int, allocator<int>
>::value_type' sugar
        |-TypeAlias 0xdb39e38 'value_type'
        `-SubstTemplateTypeParmType 0xdb39910 'int' sugar
        |-TemplateTypeParmType 0xdb09a80 'T' dependent depth 0 index 0
        | `-TemplateTypeParm 0xdb09a38 'T'
        `-BuiltinType 0xdacd950 'int'
```

... we'd wrap that in a MemberSugarType, and desugaring the wrapped function type would produce a function type whose parameter and return types are wrapped. But then we can do something more: once we get to a MemberSugarType wrapped around a SubstTemplateTypeParmType, we can desugar that type to "Foo" (taking the type sugar for the template argument from the TemplateSpecializationType). So we can desugar the type of a call to x.front() to Foo.

You'd need to work out exactly what kinds of sugar type nodes to add, and which information it makes sense to track in the sugar type. For instance, it might make sense for the sugar type that we use for a class member access to track whether the member was found in the type of the object expression (the expression to the left of the . or ->), or if it was found in a base class (and if so, which one).

Timeline

Week	Description
Community bonding period (May 20 - June 12)	
May 20 - June 12	Read documentation, get up to speed to begin working on their project. Try to frontload some of the work which requires community interaction and consensus.
Phase I (June 13 - July 29)	
1	Investigate design approaches. **Deliverable**: A concise technical document sent as a RFC to clang's developer mailing lists
2	Improve the representation of type information of a ClassTemplateSpecializationDecl. **Deliverable**: Design a new type in clang `MemberSugarType` capable of tracking type sugar
3	Improve the representation of type information of a ClassTemplateSpecializationDecl. **Deliverable:** Connect the

	MemberSugarType with the ClassTemplateSpecializationDecl.
4	Extend the test case coverage and demonstrate better diagnostics in several cases. **Deliverable:** Enhanced test cases and improved documentation where required
5	Buffer week if the review process or code development take longer than foreseen
Phase II (July 25 - September 12)	
6	Implement a single-step desugaring based on desguaring a MemberSugarType wrapped around a FunctionProtoType. **Deliverable**: The ability to push the member sugar onto the return type and parameter types of the type.
7	Implementing a MemberSugarType wrapped around a RecordType representing a member of the type tracked by a MemberSugarType would form an ElaboratedType describing the name of that member as a member of the sugared type. **Deliverable:** The ability to push the member sugar via ElaboratedType.
8	Enable MemberSugarType wrapped around a SubstTemplateTypeParmType, we can desugar that type to "Foo" (taking the type sugar for the template argument from the TemplateSpecializationType). **Deliverable:** we can desugar the type of a call to x.front() to Foo.
9	Buffer week
10	Demonstrate the usefulness of the diagnostic and also preserving of type sugar in the AST for I/O purposes (via the CERN Data Analysis Project ROOT). **Deliverable**: Remove the custom patch here.
11	Update documentation and prepare a blogpost. Fix bugs. **Deliverable:** A blog post on blog.llvm.org
12	Prepare a presentation and final report.

Personal Information

- Name: Matheus Izvekov

- Email: mizvekov@gmail.com
- Mobile number: +31 6 82412812
- Timezone: CEST (UTC+2)
- Github: mizvekov
- Residency: Amsterdam, Netherlands

Motivation & Technical Experience

I have been improving clang type sugar preservation for some time and this is one of the substantial missing pieces.

I have implemented a reasonable amount of related features and fixes to clang. For reference see:

- https://github.com/llvm/llvm-project/commits?author=mizvekov
- https://reviews.llvm.org/p/mizvekov/

The most relevant features are the ones related to type sugar, such as:

- https://reviews.llvm.org/D110216 (merged)
- https://reviews.llvm.org/D111283 (review)
- https://reviews.llvm.org/D111509 (review)
- https://reviews.llvm.org/D112374 (review)

The above features all have synergy with the current proposal, as they enable other constructs to preserve type sugar allowing both for their representation when present in the specialization argument, and in making sure we don't lose in one of the following steps this type sugar we preserved here.