Final Report

GSOC-22

Improve Clang Diagnostics

Mentors:

- **Aaron Ballman**
- **Erich Keane**
- **Shivam Gupta**

Contributor:

Muhammad Usman Shahid

codesbyusman@gmail.com

GitHub

Table of Contents

Introduction:

Error handling is an important topic in the field of computer science. The warnings, errors, and notes diagnostics on incorrect statements help the programmer to fix bugs in their code. It is an important feature of the compiler. Good diagnostics can have a significant impact on the compiler's user experience and improve productivity.

There is always significant headroom for improvement as the things are evolving, and new features are included which ultimately requires updating and adding diagnostics. This was a very impactful and interesting project to look into the preexisting diagnostics in Clang, how they can be improved, and how to add new diagnostics.

Objectives:

- Understanding clang's internal infrastructure.
- Understanding and learning the whole workflow of how a diagnostic is fired.
- Getting to know the contribution workflow forclang.
- Improving the existing diagnostics with deficient wording.
- Adding new diagnostics to clang.
- Looking into the GitHub issues and trying to resolve them.
- Interacting with the community and becoming a lifetime contributor to clang.

Motivation:

Diagnostic is dependably something critical. Diagnostic need an eye to keep them updated as new features are introduced in a language and there is always a need for an update to acknowledge the changes. The main motivation behind the project was to improve the clang diagnostic to improve error handling, user experience, and productivity

Work:

- ### Rewording Static Assertion:

I will like to divide this patch into two parts:

a) Rewording

Static assertions are the statements that perform compile-time assertion checking. For more information on the feature, please see the link provided at the end of this section. C and C++have different syntax for this feature. Consider a piece of C code:

```
_Static_assert(0, "oh no!");
```

Clang was diagnosing it as:

```
error: static_assert failed: oh no!
_Static_assert(0, "oh no!");
^              ~
1 error generated.
Compiler returned: 1
```

Here the problem was that **error: static_assert failed** which does not match the syntax written in the source code. Thus this part of the patch was focusing on rewording the diagnostic to use more general terminology. We wanted to reword the diagnostic to:

```
error: static assertion failed: "oh no!"
    1 | _Static_assert(0, "oh no!");
      | ^~~~~~~~~~~~~~
Compiler returned: 1
```

First, I just changed the diagnostic in the **DiagnosticSemaKinds.td** so that it uses the "static assertion" wording. Then the main part was the testing. I looked at the Clang test suite for all relevant test cases and changed the expected error to the desired one so that the test should pass with the new wording.

b) Parsing of used token

The **err_expected_semi_after_static_assert** diagnostic ID is used to issue the missing semicolon diagnostic after encountering a static assertion declaration. Previously, it said:

expected ';' after static_assert

which again was not considering the syntax used in the source code. Instead, it should say

- **expected ';' after 'static_assert'**
- **expected ';' after '_Static_assert'**

based on token (synta) used. Thus, I made changes to the **err_expected_semi_after_assert** diagnostic in **DiagnosticParseKinds.td** . Then I implemented the logic in **ParseDeclCXX.cpp** to inspect the token that was used and issue the correct diagnostic wording based on that.

Then I added test cases to ensure the changes are tested and are not broken in the future due to other changes. Some discussion for this can be seen from the link to the patch below, on phabricator (the code reviewing tool used by the Clang community).

Review: https://reviews.llvm.org/D129048

Commit:

https://github.com/llvm/llvm-project/commit/76476efd68951907a94def92b2bb6ba6e32ca5b4

This was the first diagnostic that I worked on improving. It was the most challenging part because it was the first time I was making changes and understanding how the internal compiler details work in practice. Initially, after testing all of the changes, the patch was ready to go. We committed the patch but the community's post commit test bots found errors that were not caught with local testing. I looked into it and started working on fixing some tests in the libc++ that needed to be updated. This also gave me a detailed view of the testing infrastructure. As a first diagnostic, it was a great experience to learn how diagnostics work, how to make changes to the compiler, and then how to write test cases for the change in diagnostic phrasing. Changing this diagnostic helped me to understand every bit of clang's diagnostic and testing infrastructure.

Reference:

- https://en.cppreference.com/w/cpp/language/static_assert
- https://www.geeksforgeeks.org/understanding-static_assert-c-11/
- https://en.cppreference.com/w/c/language/_Static_assert

Missing tautological compare warnings due to unary operators

A tautological comparison is one that is always true or always false. Clang was not diagnosing a few such statements that have a unary operator when there is a bitwise comparison. Consider the piece of code:

```
if ((x & 8) != 4) {}
```

this is always a true statement and Clang currently warns for this, but it was not warning for a statement having the unary – operator as in:

```
if ((x & -8) != 4) {}
```

The above statement is also always true but Clang was not issuing a warning for it. The problem here was that the unary operator prevented the statements from being analyzed. So I used some built-in operations to constant evaluate the integers after getting the unary operator subexpression, but that led to issuing some false positives.

The next approach I followed was to make a helper function that takes the expression as input and evaluates the given expression; (`getIntegerLiteralSubexpressionValue()`). The return type for this is `Optional<llvm::APInt>` which means that the value returned is optional; if the given expression cannot be evaluated, then the returned result has no integer value. Thus, the calling function can check whether a valid value is returned or not.

In this helper function, I first look for the unary operator, and if one is present, I dig down more into it to find its subexpression; if it is an integer literal, I calculate the value manually. By manually I mean we get the opcode used to evaluate it and return the computed value. Then I wrote several different test cases, including ones which give a tautological comparison warning and some that ensure that future changes do not regress the code.

This diagnostic was a great experience for learning about the CFG (control flow graph), and how Clang uses it when performing a diagnostic analysis. Furthermore, the unique test cases raised by the community gave me a different perspective of thinking. Although I tried to think out of the box while writing the test cases, some of the test cases given by the community made me rethink different dimensions thus, resulting in professional and educational growth.

Review: https://reviews.llvm.org/D130510

Commit:

https://github.com/llvm/llvm-project/commit/fd874e5fb119e1d9f427a299ffa5bbabaeba9455

GitHub Issue: https://github.com/llvm/llvm-project/issues/42918

- ## Diagnosing the Future Keywords

In programming languages, there are many reserved keywords. For example, **int** is a reserved keyword for declaring the integer variables. Clang has existing diagnostics for keywords, but as new versions of languages are released, new keywords are added and the diagnostics were not updated. Thus changes were needed. Furthermore, the diagnostic was improved to also diagnose when something will be a keyword in a future language mode. This warning is important so that users know when an upcoming version will have a keyword which will break the user's code.

In some versions of C, we don't have bool, true, or false keywords. You are allowed to make declaration such as:

int bool;

It will not be an error before C2x(C23), but in C23, bool, true, and false are keywords. In that language mode, the code will have an error because it uses a reserved keyword. After these changes, when I declare **int bool;** a warning is generated as:

'bool' is a keyword in C2x

Thus, these are now diagnosed as a future reserved keyword so users know not to use that identifier if they intend to upgrade to a newer language standard. We made these changes for both C and C++ keywords.

For this, we defined the tokens as being keywords in a particular language mode in **TokenKinds.def.** This file defines the TokenKind database. It is a kind of database for the various operators and keywords. There is a token KS_Future that is set for the specific keyword to make it future by mapping how they are defined in the **TokenKinds.def**. This process was very confusing and mixed up so in between Erich gave it a new look, and logic and it was easy to understand easily, especially for me. This revision contains that changes:

⚙ D131007 [NFCI] Refactor how KeywordStatus is calculated (llvm.org)

After these clarifying changes, I called helper functions with the necessary information to achieve the goal. Finally, the most important thing was the test cases written to ensure the keywords and the warning they emit were correct. Furthermore, it ensures that if someone makes invalid changes then the test fails.

This diagnostics was a completely different experience for me. Like all other diagnostics, it was also one from which I learned many things. The way Erich changed calculating the status of the keyword was again for me a learning process that ultimately resulted in my growth. I now understand preprocessing better and how keywords are formed while lexing. Although, as a complete beginner, I had to look into it more deeply and may have taken more time, but it was a great experience for learning about keywords.

Review: https://reviews.llvm.org/D131683

Commit:
https://github.com/llvm/llvm-project/commit/41667a8b9b624e282e7c08fadf7091223728d1c1

- ## Reinterpret_cast

The diagnostic here was correct but its wording was really silly and confusing. It was talking about `reinterpret_cast` in C and our goal was to find better wording for the C language. Thus we modified the **note_constexpr_invalid_cast** diagnostic**.**

Mainly we focused on the `%select{}` string format. It is a handy approach, I will say. Instead of writing separate diagnostics for minor changes in wording, it allows you to select between several options within the same diagnostic. Let say I have to say:

My name is Usman.

And in place of Usman, I have to use multiple names, instead of writing multiple lines I can achieve this with a single line as in:

My name is %select{usman|muhammad|shahid}0.

When issuing the diagnostic, the value of the first streamed argument will determine what is printed. If zero is passed, usman will be printed, if one is passed, muhammad will be printed, etc.

So we introduced a new select statement in the **note_constexpr_invalid_cast** as in:

%select{this conversion|cast that performs the conversions of a reinterpret_cast}1

So this statement depends on the second streamed argument passed, which is the current language mode. If the language mode is C++ it returns 1, which will result in:

cast that performs the conversions of a reinterpret_cast

otherwise, if it is in C mode then it results in:

this conversion

From the example:

int array[(long)(char *)0];

the previous note was:

cast that performs the conversions of a reinterpret_cast is not allowed in a constant expression

and the reworded note is now:

this conversion is not allowed in a constant expression

Finally, I worked on the tests, and it was again a learning experience. Each time I spent in the GSoC for me was a growth time whether it was educational, professional, or personal.

Review: https://reviews.llvm.org/D133194

Quick Links:

a) GitHub commits:

- https://github.com/llvm/llvm-project/commits?author=Codesbyusman

b) Reviews:

- https://reviews.llvm.org/D129048
- https://reviews.llvm.org/D130510
- https://reviews.llvm.org/D131683
- https://reviews.llvm.org/D133194

Future work:

After understanding the flow and the internal infrastructure and contributing mechanism, I am willing to be a permanent contributor to Clang. I will continue exploring Clang's behavior and GitHub issues related to diagnostics and solve them with the help of the community and knowledge I gain.

Experience:

Like most other undergraduate students, I also don't have specific practical work. I found the GSOC program to be a great platform to learn new technologies, sharpen my skills, and have practical work experience based on the theoretical knowledge that I gained from my degree. From the start, open source fascinates me, as the whole community works together. Thus, at the time, I found the project Improvement of Clang Diagnostics that needs knowledge in C++. I have been studying C++ for the last 2 years, so when I saw that project , I wrote a proposal that I understood at that time and started exploring Clang.

After the proposal was accepted, I started looking deeper into Clang. I understood the basic structure and started learning how to contribute to Clang and configure the whole setup. At the start, I explored things including the testing infrastructure of Clang. I practiced writing test cases

for a real system. With the help of my mentors, I started to work on diagnostics and with their never-ending guidance, I successfully achieved my goals.

It was a great time for me. I have learned very different things. I got some new coding styles and different logic-building techniques. This experience has increased my knowledge and my skill set.

Acknowledgment:

I would like to thank the whole community for being there and suggesting solutions. Especially my mentors Aaron Ballman, Eric Keane, and Shivam Gupta who were always there to help me, guide me, and support me. They were a great motivation to keep me going. I would like to add that the way they assigned me diagnostics was impressive. I felt that they were teaching me step by step to higher and more complex things, diving me deeper and deeper into the project. This was the most impressive thing I had experienced. They helped me tirelessly during this time by quick responses to my emails, reviewing my patches, and giving advice.

Conclusion:

These months are the most memorable and cherished moments of my life. Over time, I have learned many new things that have sharpened my skills. It was a great time of learning, coding, hacking on Clang, struggling, and interacting with the community. Now with great confidence, I can say that after this GSOC program I can contribute to Clang with help on diagnostics.