Compiler Engineering
CS6383 - Sep 2019
Mini Assignment 2:

Error Messages in Clang, GCC, and LLVM

INSTRUCTOR: Dr. Ramakrishna Upadrasta

Report By:
Vijay Tadikamalla - CS17BTECH11040
Tungadri Mandal - CS17BTECH11043

भारतीय प्रौद्योगिकी संस्थान हैदराबाद
Indian Institute of Technology Hyderabad

Error Messages In Clang

Error messages thrown by Clang are extremely user-friendly. These messages are written to stderr(the standard error file) and they have the following form.

```
$ clang -fsyntax-only format-strings.c
format-strings.c:91:13: warning: '.*' specified field precision is missing a matching 'int' argument
  printf("%.*d");
            ^
```

Here *format-strings.c* is the filename. The integers after the file name are corresponding to the line number in the file and the corresponding column from where the error originated.

Clang has a class DiagnosticsEngine to report problems and issues (located in ../include/clang/Basic/Diagnostic.h file). Its primary purpose is to provide a way to report a diagnostic via its Report method, i.e by passing its diagnostics to the DiagnosticsConsumer for reporting it to the users.

Each diagnostic is constructed by the DiagnosticsEngine::Report method which also allows insertion of some extra information like **arguments and source ranges** into the diagnostic. According to the severity and type of the Diagnostic, clang classifies them into different levels:

```
// The level of the diagnostic
enum Level {
  Ignored = DiagnosticIDs::Ignored,
  Note = DiagnosticIDs::Note,
  Remark = DiagnosticIDs::Remark,
  Warning = DiagnosticIDs::Warning,
  Error = DiagnosticIDs::Error,
  Fatal = DiagnosticIDs::Fatal
};
```

Another part of clang's diagnostics is the class DiagnosticsBuilder. This is a helper class that is used to produce diagnostics.

Special features/properties of Error messages thrown by Clang

- **Sources Ranges** is a special feature in clang diagnostics. It helps to highlight a particular part of the code which is a plausible source of error. For example:

```
// file.c
void f();
int x = 5 + f;
```

```
> $ clang file.c
file.c:3:13: error: invalid operands to binary expression ('const
char *' and 'void (*)()')
int x = "5" + f;
        ~~~ ^ ~
1 error generated.
```

- We use high-level user-defined types, typedefs, and other syntactic sugar to refer to types in our programs. However, sometimes it is important to strip off the typedef to understand what is going on. So, clang exposes the underlying details of a typedef.
  For example, if the user was confused about how the system ***"pid_t"*** typedef is defined, Clang helpfully displays it with ***"aka"***.

```
> $ clang -fsyntax-only t.c
t.c:13:9: error: member reference base type 'pid_t' (aka 'int') is
not a structure or union
    myvar = myvar.x;
```

- Clang also throws some warnings/remarks, also known as ***"Fix-it"*** hints provide advice for fixing small, localized problems in source code like non-standard or redundant syntax, missing keywords, common mistakes, etc.
  For example, it gives us warnings if we use old/deprecated syntax.

```
> $ clang t.c
  t.c:5:28: warning: use of GNU old-style field designator extension
  struct point origin = { x: 0.0, y: 0.0 };
                          ~~ ^
                          .x =
```

Error Messages in LLVM

According to the LLVM documentation:

> Proper error handling helps us identify bugs in our code, and helps end-users understand errors in their tool usage. Errors fall into two broad categories: *programmatic* and *recoverable*, with different strategies for handling and reporting.

Programmatic errors are violations of program invariants or API contracts and represent bugs within the program itself. Primary tools to handle them are assertions and llvm_unreachable which are explained below.

Recoverable errors represent an error in the environment itself like missing files or resources and other such types of failures. These should be detected and reported to a level where they can be handled properly. Some may be just reporting to the user and the other may be trying to recover from the error itself.

> Recoverable errors are modeled using LLVM's `Error` scheme. This scheme represents errors using function return values, similar to classic C integer error codes, or C++'s `std::error_code`. However, the `Error` class is actually a lightweight wrapper for user-defined error types, allowing arbitrary information to be attached to describe the error. This is similar to the way C++ exceptions allow throwing of user-defined types.
>
> Success values are created by calling `Error::success()`, E.g.:

```
Error foo() {
  // Do something.
  // Return success.
  return Error::success();
}
```

There are many ways to display meaningful errors in LLVM. One such way is the Error Class Reference. This is a lightweight error class with error checking and mandatory checking.

Another way to check is by using the assert macro which is used for programmatic errors. It is encouraged in the LLVM coding standards to use the LLVM assert macro to its full potential to check all preconditions and assumptions as we may never know when a may get detected and it may significantly reduce the debugging time. We also put an error message in the statement itself which will be shown if the assertion is activated. It makes the error messages richer and lets the debugger know which assertion his or her program failed and why.

Initially, the *asserts* were used for codes that should not be reached. But the problem was that the compiler may not be able to understand this or warn about a missing

return in builds where assertions are compiled out. But now we have *llvm_unreachable* to handle this. When assertions are disabled it automatically does not generate code for this branch.

But neither assertions or *llvm_unreachable* may be used in the final release builds. If they are somehow triggered by the users then error handling has to be done or *report_fatal_error* can be used.

Another important thing to note that values used by assertions will give unused value when assertions are disabled. As per the documentation :

> These are two interesting different cases. In the first case, the call to `V.size()` is only useful for the assert, and we don't want it executed when assertions are disabled. Code like this should move the call into the assert itself. In the second case, the side effects of the call must happen whether the assert is enabled or not. In this case, the value should be cast to void to disable the warning. To be specific, it is preferred to write the code like this:

```
assert(V.size() > 42 && "Vector smaller than it should be");

bool NewToSet = Myset.insert(Value); (void)NewToSet;
assert(NewToSet && "The value shouldn't be in the set yet");
```

Error Messages In GCC

The GCC compiler can produce two kinds of diagnostics: errors and warnings. They both have a different purpose. According to the official documentation:

> *Errors* report problems that make it impossible to compile your program. GCC reports errors with the source file name and line number where the problem is apparent.
> *Warnings* report other unusual conditions in your code that *may* indicate a problem, although compilation can (and does) proceed. Warning messages also report the source file name and line number, but include the text '`warning:`' to distinguish them from error messages.

Warnings generally refer to problematic points where the program may not function as intended or there may be the use of some deprecated features in those points or perhaps even the use of non-standard features of GCC or G++. Many warnings are issued only if they are explicitly requested with one of the -W options. One important thing to note about GCC according to its original documentation is as follows:

> GCC always tries to compile your program if possible; it never gratuitously rejects a program whose meaning is clear merely because (for instance) it fails to conform to a standard. In some cases, however, the C and C++ standards specify that certain extensions are forbidden, and a diagnostic *must* be issued by a conforming compiler. The `-pedantic` option tells GCC to issue warnings in such cases; `-pedantic-errors` says to make them errors instead. This does not mean that *all* non-ISO constructs get warnings or errors.

Error messages in GCC are written in a format called '***brief-format***'. These messages are written to stderr(the standard error file) and they have the following form.

```
e.adb:3:04: Incorrect spelling of keyword "function"
e.adb:4:20: ";" should be "is"
```

Here e.adb is the filename. The integers after the file name are corresponding to the line number in the file and the corresponding column from where the error originated. Please note that this is not the actual error message but it tells you where the error originated from and possible suggestions to remedy the error.

One of the ways in which GCC generates errors is through the directive ***'#error'*** which causes the preprocessor to report a fatal error. The following tokens after this directive are taken as the error message. For example:

You would use '`#error`' inside of a conditional that detects a combination of parameters which you know the program does not properly support. For example, if you know that the program will not run properly on a VAX, you might write

```
#ifdef __vax__
#error "Won't work on VAXen.  See comments at get_last_object."
#endif
```

If you have several configuration parameters that must be set up by the installation in a consistent way, you can use conditionals to detect an inconsistency and report it with '`#error`'. For example,

```
#if !defined(FOO) && defined(BAR)
#error "BAR requires FOO."
#endif
```

This is for user-generated errors. Similarly, for the warning, we can use the directive ***'#warning'***, which just issues a warning and continues preprocessing. Similar to the error directive the warning directive also takes the following tokens as warning messages.

Before going to how the errors are reported one thing we can do in GCC is :

GCC allows the user to selectively enable or disable certain types of diagnostics, and change the kind of the diagnostic. For example, a project's policy might require that all sources compile with `-Werror` but certain files might have exceptions allowing specific types of warnings. Or, a project might selectively enable diagnostics and treat them as errors depending on which preprocessor macros are defined.

Errors are reported through the following files. In the GCC subfolder of the GCC repository, there exist two files. One is diagnostic.c and the other is error.c. The diagnostic.c file contains the language-independent diagnostic subroutines for the GCC. The errors.c provides basic error reporting routines. These 2 are the main files responsible for error reporting.

Each language-related sub-folder in this folder (../gcc) have their own errors.c. For example, the c folder (../gcc/c) has a file called c-errors.c which provides diagnostic subroutines for GNU c language.

Similarly, for go folder there exists a folder called gofrontend (../gcc/go/gofrontend). It has a file called go-diagnostics.h.

The frontend is written in C++. It can only be used in conjunction with a full compiler backend. Currently the backend interface has been implemented with GCC (known as gccgo) and with LLVM (known as GoLLVM). The backend is expected to provide the functions defined at the bottom of go-diagnostics.h: go_be_error_at, etc. These will be

used for error messages. These go-diagnostics.h is the interface to diagnostic reporting.

Again similarly, in the Fortran folder (../gcc/fortran) there is error.c which handles errors.

Now the error handlers for the libraries of the languages are in their library folders in the root directory and are not located in the GCC directory. For example for c++, its error handler for CPP library is in the libcpp folder (../libcpp) called errors.c.

References

1. https://clang.llvm.org/diagnostics.html
2. http://www.goldsborough.me/c++/clang/llvm/tools/2017/02/24/00-00-06-emitting_dia
3. gnostics_and_fixithints_in_clang_tools/
4. https://www.learncpp.com/cpp-tutorial/configuring-your-compiler-warning-and-error-levels/
5. https://github.com/gcc-mirror/gcc
6. https://gcc.gnu.org/onlinedocs/