Compilers Engineering 2019
Mini Assignment 2:
Error messages and their flow in GCC, Clang and LLVM
Report

Sai Harsha Kottapalli
CS17BTECH11036

September 9, 2019

1 GCC

On compilation of source code, gcc usually produces two types of output if it has failed.
One being "errors" and other being "warnings".
Errors are shown in the format:

```
bubbleSort(arr, n)

bubblesort.cpp: In function 'int main()':
bubblesort.cpp:38:23: error: expected ';' before 'cout'
```

where the file's name, scope(limited to functions), line number, column number and the problem occurred while compilation is shown.
In above example we forgot to put a semi-colon after the statement which gave rise to the associated error.

In the case of Warnings, they are usually the code which might still run during execution but is not what the user might desire.

```
int swap(int *a, int *b) {
        int temp = *a;
        *a = *b;
        *b = temp;
}

bubblesort.cpp: In function 'int swap(int*, int*)':
bubblesort.cpp:9:1: warning: no return statement in function returning
non-void [-Wreturn-type]
```

Here, though the syntax is same as that of error it always has "**warning:**" to differentiate from other errors.
Usually most warnings help us find bugs without actually having to debug the entire code to identify it and also helps in writing a dis-ambiguous code.

However internally in gcc, there are more cases: warning, error, fatal, pedantic warnings, etc. In gcc/diagnostic.c and gcc/diagnostic-core.h, The definitions are defined where:
Warnings: "correct according to a language specification but likely to be buggy".
Error: "ill-formed code"
Fatal: "An error which is severe enough that we make no attempt to continue."
This file implements "the language independent aspect of diagnostic message module" based on the flags passed for compilation.

warning function just prints warning in the discussed format and continues.
Error function, prints the error and has a flag set to 1 (have_error) so that the object file is not generated.
Fatal function, just quits at that point which the associated error message and code.

These functions are invoked from series of functions in gcc not necessarily in same file which might be why location_t is an argument.

2 LLVM/Clang

In LLVM/Clang, errors are primarily divided into two types: programmatic and recoverable errors.

2.1 Programmatic Errors

These arise when there are fatal bugs in the source code where the program invariants are not satisfied.
The course of action in this case is to just quit at the point of failure.
The invariant conditions are checked through the help of assert.
Example:
assert(V1.getType() == V2.getType && "Constant types must be identical!").
Usually without this check, the program would assume everything is correct and execute and debugging it to this case takes time.

2.2 Recoverable Errors

As the name suggests, these errors might be possible to be recovered.
This includes cases where a required file is missing, etc.
Usually these are notified to the user as well.
We can also call report_fatal_error which works just like fatal function defined in gcc.
These errors use the Error scheme from LLVM. Success values (return type) is invoked with Error::success() while failure values with make_$< T >$, given T is an class extension of ErrorInfo.
Some files for Error Handling in clang:

- ErrorHandling.cpp
  Primarily for callbacks for errors. The files describes itself as - "defines an API used to indicate fatal error conditions".

- LLVMContext.cpp
  Most of the non-fatal errors are handled here.

- errno.h
  List of Error codes and their definitions.

- ErrorOr.h
  Provides ErrorOr¡T¿ smart pointer.

FLOW:
Parser finds that there is an error.
c_parser_set_error is responsible for sending error details. This is passed as an object with associated message and it's location (location_t).
error function prints it to stderr.

References: llvm.org/docs/ProgrammersManual.html