Compiler Errors | Coursera

Compiler Errors

Compiler Errors

The compiler is a rather complex program. Its first task is to read in your program source and “understand” your program according to the rules of C—a task called the program. In order for the compiler to parse your program, the source code must have the correct syntax (as we discussed in Course 1). If your code is not syntactically correct, the compiler will print an error message and attempt to continue *parsing*, but may be confused by the earlier errors. For example, the following code has one error—a missing semi-colon after int x = 3 on line 5:

```
9

6

7

8

3

4

5

1

2

}

  int y = 4;

  printf("%d\n", x + y);

  return EXIT_SUCCESS;


int main(void) {

  int x = 3

#include <stdio.h>

#include <stdlib.h>
```

However, compiling the code results in two errors:

```
err1.c: In function 'main':
err1.c:6: error: expected `,' or `;' before `int'
err1.c:7: error: `y' undeclared (first use in this function)
err1.c:7: error: (Each undeclared identifier is reported only once
err1.c:7: error: for each function it appears in.)
```

The first error (expected `,' or `;' before `int') correctly identifies the problem with our code—we are missing a semi-colon before the compiler sees int at the start of line 6 (it should go at the end of line 5, but the compiler does not know anything is wrong until it sees the next word). The next error ('y' undeclared) is not actually another problem with the code, but rather a result of the compiler being confused by the first error—the missing semicolon caused it to misunderstand the declaration of y as a part of the previous statement, so it does not recognize y later in the code.

Compiler errors can be a source of confusion and frustration for novice programmers. One key rule in dealing with them is to try to fix them in order, from first to last. If you find an error message other than the first one confusing, you fix the first error, then retry compiling the code and see if the error goes away—it may just be a result of the compiler being confused by earlier errors.

Dealing With Compilation Errors: Tip 1 Remember that the compiler can get confused by earlier errors. If later errors are confusing, fix the first error, then try to recompile before you attempt to fix them.

Another source of confusion in dealing with compiler errors is that the error message may not do a good job of describing the problem. The compiler tries its best to describe the problem, but may guess incorrectly about what you were trying to do, or refer to unfamiliar possibilities. For example, if we have a slightly different variation on the missing semicolon problem from the previous example:

```
1  #include <stdio.h>
2  #include <stdlib.h>
3  
4  int main(void) {
5    int x
6    int y;
7    x = 3;
8    y = 4;
9    printf("%d\n", x + y);
10   return EXIT_SUCCESS;
11 }
```

The error here is still a missing semicolon after the declaration of x on line 5, however, the compiler gives the following error messages:

```
err2.c: In function 'main':

err2.c:6: error: nested functions are disabled, use -fnested-
functions to re-enable

err2.c:6: error: expected `=', `,', `;', `asm' or `__attribute__' before `int'

err2.c:7: error: `x' undeclared (first use in this function)

err2.c:7: error: (Each undeclared identifier is reported only once

err2.c:7: error: for each function it appears in.)
```

These error messages may appear quite intimidating, since they refer to several language features which we have not seen (and which are all irrelevant to the problem). The first error message here is actually completely irrelevant, other than that it is on the right line number. The compiler first guesses that we might be trying to use a non-standard language extension (extra features that one particular compiler allows in a language) that allows us to write one function inside of another. This extension is disabled by default, and the compiler thinks that we might want to enable it—it is trying to be helpful, even though it is wrong. In general, if an error message is referencing something completely unfamiliar, it is likely that the compiler is confused, or offering suggestions that are not relevant to what you are trying to do.

Dealing With Compilation Errors: Tip 2 If parts of an error message are completely unfamiliar, try to ignore them and see if the rest of the error message(s) make sense. If so, try to use the part that makes sense to understand and fix your error. If not, search for the confusing parts on Google and see if any of them are relevant.

Looking at the next error message, we see that it is on the same line number (6), so the compiler may be trying to give us more or different information about the same problem. Here, while there are some unfamiliar things (such as asm or __attribute__), the rest of the error message makes sense if we ignore these. That is, we could understand the error message if it were:

```
err2.c:6: error: expected `=', `,', or `;' before `int'
```

The compiler has told us that it expects an equals sign, comma, or semicolon before we get to the word **int** on line 6. In this case, we are missing a semicolon, but the compiler is suggesting other possibilities (e.g., we could have had an = had we been writing **int** x = 3;). The asm and __attribute__ suggestions it provided could also have been options, but are very advanced language features which we will not cover.

Another source of confusion in dealing with error messages can arise from the fact that the compiler may not notice the error immediately. As long as the source code seems to obey the rules of C’s syntax, the compiler assumes everything is correct, even if its parse does not agree with what you meant. In the following code, the actual error is omission of a close curly brace on line 9:

```
1

2

3

4

5

6

7

8

9

10

11

#include <stdio.h>

#include <stdlib.h>


int main(void) {

  for (int i = 0; i < 10; i++) {

    for (int j = 0; j < 10; j++) {

      printf("%d\n", (i+3) * (j+4));

    }
  return EXIT_SUCCESS;
}
```

However, the compiler does not recognize the error immediately. Having a return statement inside the for loop is syntactically valid. The closing brace after it on line 11—which the programmer intended to be the end of main is also legal, but the compiler parses it as the end of the first for loop. The compiler then discovers that something is not right when it reaches the end of the file and the body of main has not ended, reporting the error as:

```
err3.c: In function 'main':
err3.c:11: error: expected declaration or statement at end of input
```

Dealing With Compilation Errors: Tip 3 Programmer's editors are very good at helping you find mismatched braces and parenthesis. Most will indent the code according to the nesting level of the braces it is inside, and will show you the matching brace/parenthesis when your cursor is on one of the pair. Using these sorts of features can be the easiest way to find errors from mismatched braces and parenthesis.

Once the compiler has parsed your program, it type-checks the program. Type-checking the program involves determining the type of every expression (including the sub-expressions inside of it), and making sure that they are compatible with the ways that they are being used. It also involves checking that functions have the right number and type of arguments passed to them. The compiler will produce error messages for any problems it discovers during this process.

Another important note about fixing compiler errors is that sometimes fixing one error will let the compiler progress further along, resulting in more errors. Novice programmers who are not confident in their correction of their first error may undo the fix, thinking they made things worse, even though the fix may have been correct—the extra errors just came from progressing further through the process.

Dealing With Compilation Errors: Tip 4 Be confident in your fix for an error. If you do not understand what is wrong and how to fix it, find out and be sure rather than randomly changing things.

http://aop.cs.cornell.edu/errors/index.html provides more details on a variety of error messages, and will likely prove useful in helping you diagnose errors you may encounter.

Once the compiler finishes checking your code for errors, it will translate it into assembly—the individual machine instructions required to do what your program says. You can ask the compiler to *optimize* your code—transform it to make the resulting assembly run faster—by specifying the -O option followed by the level of optimization that you want. Programs are usually compiled with no optimizations for testing and debugging (the code transformation makes the program very difficult to debug), and then re-compiled with optimizations once they are ready for release/use. Typically, real programs compiled with gcc are compiled with -O3 when they are optimized. gcc provides a variety of other options to control optimization at a finer level of detail, however, they are beyond the scope of this specialization. We are strictly concerned with writing code that works, so you do not need to bother with the -O flag for any of the exercises you do in this specialization.

✓

Completed