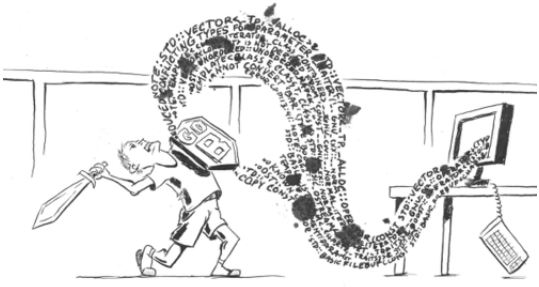
C in Practice

Kaiyuan Xu

March 1, 2021

Pitfall: Write a very long piece of code and compile (test) it afterwards.

From one extreme to the other: test driven development.



Pitfall: Ignoring warnings.

Always use `-Wall -Wextra` flags.

Fallacy: The code works (compiles) will always work (compile).

So many things could go wrong!

- problematic header files
- symbol conflicts
- linkage errors
- runtime errors
- ...

Declaration Vs. Definition.

You can declare a function/structure as many time as you want (exactly the same), but you can only define a function/structure once.

```
1 int fn(); /* declaration */
2
3 int fn() { return 0; } /* definition */
4
5 struct a; /* declaration */
6
7 struct a {
8     int first;
9 }; /* definition */
```

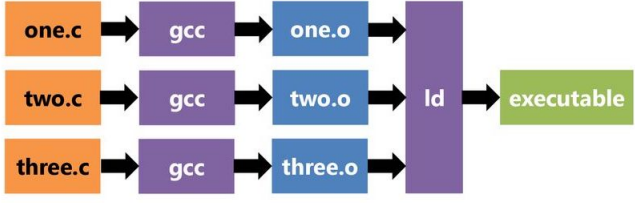
Linkage Error: Why it happens?

```
/usr/bin/ld: b.o: in function `fn_a':
b.c:(.text+0x0): multiple definition of `fn_a';
a.o:a.c:(.text+0x0): first defined here
/usr/bin/ld: a.o: in function `main':
a.c:(.text+0x52): undefined reference to `var_c'
collect2: error: ld returned 1 exit status
```

Translation Unit

A **translation unit** is the ultimate input to the compiler from which an object file is generated.

Compiler can only see part of the program, which means some error can only be discovered at link time.



Object File: Where is my symbol?

```c
int fn_a(void) {
    return 0;
}
static int fn_b(void) {
    return 0;
}
extern int fn_c(void);
int fn_d(void);
int var_a = 0;
static int var_b = 0;
extern int var_c;
int var_d;
```

```
Section Headers:
  [Nr] Name              Type             Flags
  [ 1] .text             PROGBITS          AX
  [ 4] .bss              NOBITS            WA

Symbol table '.symtab':
Size Type    Bind   Vis       Ndx Name
  15 FUNC    GLOBAL DEFAULT     1 fn_a
  15 FUNC    LOCAL  DEFAULT     1 fn_b
   0 NOTYPE  GLOBAL DEFAULT   UND fn_c
   0 NOTYPE  GLOBAL DEFAULT   UND fn_d
   4 OBJECT  GLOBAL DEFAULT     4 var_a
   4 OBJECT  LOCAL  DEFAULT     4 var_b
   0 NOTYPE  GLOBAL DEFAULT   UND var_c
   4 OBJECT  GLOBAL DEFAULT   COM var_d

Relocation section '.rela.text':
  Offset          Type         Sym. Name + Addend
00000000004c  R_X86_64_PLT32    fn_c - 4
00000000005c  R_X86_64_PLT32    fn_d - 4
000000000052  R_X86_64_PC32     var_c - 4
000000000062  R_X86_64_PC32     var_d - 4
```

Global Variables Are Evil.

Linker could merge any `COMMON` global variable against any other global variable (even with different type) with the same name. Using `-fno-common` flag (default in GCC 10) can avoid generate `COMMON` global variable.

Using `extern` variable in header file is better but global variables are still bad.

Never use global variables! Define a static variable and access it through some functions instead, which can also help prevent concurrent bugs.

`#include`: Nothing magical!

Preprocessor simply replace `#include` directive with the file specified.

More effort is needed to make your header files bulletproof.

Mistake: Not using `#include` guard.

This will not necessarily cause errors, but is considered as bad practice.

```
1  #ifndef PROJECT_NAME_FILE_NAME_H
2  #define PROJECT_NAME_FILE_NAME_H
3
4
5  /*****
6   *
7   * The code here will never be 'included' twice.
8   * The #include guard should be used in any header files.
9   *
10  *****/
11
12
13 #endif /* PROJECT_NAME_FILE_NAME_H */
```

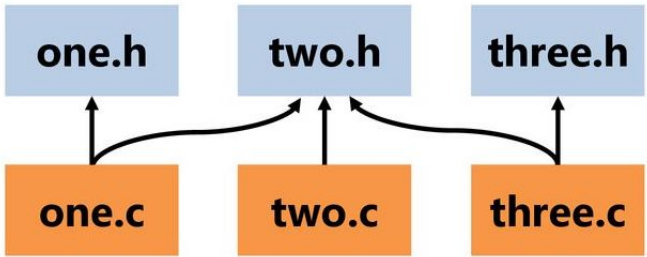
Mistake: Recursive include.

With include guard, one of the recursive include will have of effect. This could lead to some mysterious ‘undefined reference’ errors.

Reorder your header file and break recursive dependency. If that is impossible, declare the function you need instead of including it.

Mistake: Not making the header file compilable by itself.

Header file should include any dependency file, not relying on the file including it.



Mistake: Placing static variable in header file.

The same variable will be generate over and over again!

What about static functions?

`inline` is introduced in **C99**, this could prevent generating duplicate functions and even improve performance in some situation.

What should go into an header file?

Header files are used to provide information needed in different translation units. Things can be put in header files:

- macro definitions
- structure (enum, typedef) definitions (declarations)
- function declarations
- (**C99**) static inline functions which are small and simple

GDB: Locate runtime errors.

Using -g flag to generate debug information for gdb. Using command `gdb` to invoke debugger.

- `b`: set break point.
- `c`: Continues running the program until the next breakpoint or error.
- `s`: Runs the next line of the program.
- `bt`: show the current stack back trace.
- `p`: print variable.
- `info stack full`: show the current stack along with all variables on stack.

Mordern Compilers: Faster than assembly.

Nowadays compiler can generate highly optimized code probably better than hand written assembly.

- different optimization levels: `-O1`, `-O2`, `-O3`, `-Os`.
- link time optimization: `-flto`.

Q. & A.