Jakub Levý

Summary

The specialized text is an excerpt from the book *C Traps and Pitfalls* written by Andrew Koenig[1]. The text contains introductory chapter 0, the whole content of the first two chapters and a small taste of the third chapter of the book. All of the chapters are oriented towards pitfalls every C programmer even experienced one, may get into.

Chapter 1 (Lexical pitfalls) encompasses mainly the idea of techniques of lexical analysis that all C compilers implement. After reading this chapter a careful reader will be familiar with greedy lexical analysis and understand that `a---b` means the same as `a-- - b` rather than `a- --b`.

Chapter 2 (Syntactic pitfalls) In the very beginning of this chapter a reader is introduced to a formal definition of C variable declaration as it turns out to be necessary for understanding complicated C types. The chapter continues with examples of pitfalls caused by the ignorance of operator priorities. The last section wraps the chapter up with mind-blowing pitfalls caused by superfluous/missing semicolons.

Chapter 3 (Semantic pitfalls) starts with a brief overview of a notion of pointers and arrays and discusses the differences that may sometimes arise when mindlessly interchanging them.

[1] a former AT&T and Bell Labs researcher and programmer. Mostly known for his later work on C++ standards.

Opinion

The first book I read from Andrew Koenig was *Accelerated C++*[2], it was back in 2014. I immediately became fond of his writing style – write less than others but cover the same. Even though the first reading is usually a bit dense, one can learn much faster.

On closer inspection, the typesetting of the text looks rather modern, but the book was published in 1989 which is quite a lot for a book about a certain programming language. A knowledgeable reader cannot just overlook that some of the presented pitfalls are not a problem any more thanks to the standardization committee that works hard and brings a better language to us every year. For example, one pitfall was presented by `a =/*b`. Although it might look like `a = /* b`, it is actually handled by a tokenizer quite differently – it actually means `a =/ *b`. Unfortunately the last version of C language that used `/=` as an operator was so called *K&R C*[3]. Later on, C language was standardized by *ANSI*[4] and `=/` was superseded by `/=` and this pitfall vanished.

The previous paragraph indicates the presence of outdated pitfalls. Even though the book is 30 years old so far, there is still plenty of pitfalls that are still valid. My favorite is this one:

---

[2]This book basically covers the same ground as the legendary C++ Primer (1k pages) but does so on a fourth of its space. This is particularly so because it does not attempt to be an introduction to programming, but an introduction to C++ for people who've previously programmed in some other language. It has a steeper learning curve, but, for those who can cope with this, it is a very compact introduction to the language.

[3]In 1978, Brian Kernighan and Dennis Ritchie published the first edition of The C Programming Language. This book, known to C programmers as K&R, served for many years as an informal specification of the language. The version of C that it describes is commonly referred to as K&R C.

[4]ANSI = American National Standards Institute

Suppose you were given the following piece of code to audit.

```
if(flag_A & flag_B) {
   //execute code here if the condition is satisfied
   ...
}
```

This code is perfectly valid, but the condition looks mysterious to non-C programmers. It doesn't look like a proper condition even though it means:

if `flag_A` *bitwise_and*[5] `flag_B` is not equal to 0 then execute code shown as three dots.

A good idea is to make this explicit so you might rewrite your code to:

```
if(flag_A & flag_B != 0) {
   ...
}
```

This statement is now easier to understand. It is also wrong, because `!=` binds more tightly than `&`, so the interpretation is now:

```
if(flag_A & (flag_B != 0)) {
   ...
}
```

What you had in mind must be written with explicit parentheses this way:

```
if((flag_A & flag_B) != 0) {
   ...
}
```

Maybe that is the reason why the original code relied on implicit comparison with 0 and didn't use any explicit parentheses.

[5] An example of bitwise_and operation (values are in a binary representation):

`flag_A =`	1	0	0	1
`flag_B =`	1	1	0	0
`result =`	1	0	0	0

Overall this book is worth the time spent on if you consider yourself to be a C expert. Some examples are obsolete whereas others are not. I would definitely recommend it to anyone who is interested in historical problems of C language and history of programming in general, but I do not completely agree with a sentence from the preface ”This book belongs on your shelf if you are using C at all seriously even if you are an expert...”. In my opinion, you can definitely do without and spend your time on different C books.