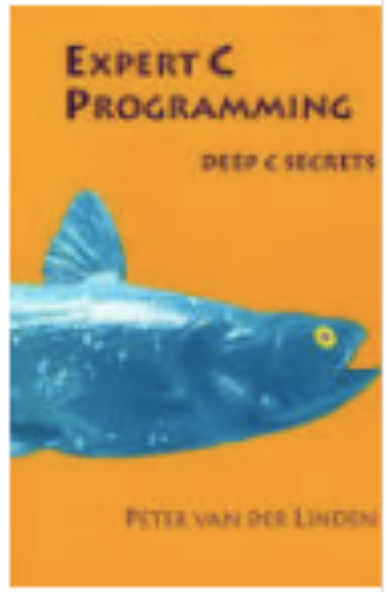
Fishing for better C programming…

Deep C Secrets

Interesting aspects of programming in C

Expert C Programming : Deep C Secrets

Book by Peter van der Linden



4.3/5 · Goodreads

This book is for the knowledgeable C programmer, this is a second book that gives the C programmers advanced tips and tricks. This book will help the C programmer reach new heights as a professional. ... Google Books

Originally published: 1994

Author: Peter van der Linden

Not all C statements should be used…

Software Dogma

The Switch Statement that defeated AT&T

The Switch Statement

- When a `break` statement is reached, the `switch` terminates, and the *flow of control* jumps to the next line following the `switch` statement.
- If no `break` appears, the *flow of control* **will fall through** to subsequent cases until a `break` is reached.

```
switch(expression) {

   case constant-expression:
      statement(s);
      break;    /* Optional */

   case constant-expression:
      statement(s);
      break;    /* Optional */

   /* You can have any number
      of case statements */

   default :    /* Optional */
      statement(s);

}
```

```
i = 2;
switch (2) {
  case 1: printf("case 1 \n");
  case 2: printf("case 2 \n");
  case 3: printf("case 3 \n");
  case 4: printf("case 4 \n");
  default: printf("default \n");
}
```

…will print out

```
case 2
case 3
case 4
default
```

This is known as "**fall through**" and was intended to allow **common** end processing to be done, after some **case-specific** preparation had occurred.

In practice it's a severe **misfeature**, as almost all case actions end with a `break;`.

This is a replica of the code that caused a major disruption of AT&T phone service throughout the U.S. AT&T's network was in large part unusable for about nine hours starting on the afternoon of January 15, 1990.

```
network code()
{
   switch (line) {
        case THING1:
          doit1();
          break;
        case THING2:
          if (x == STUFF) {
               do_first_stuff();
               if (y == OTHER_STUFF)
                    break;
               do_later_stuff();
          }                    /* Coder meant to break to here... */
          initialize_modes_pointer();
          break;
        default:
          processing();
   }                           /* But actually broke to here! */
   use_modes_pointer();        /*   leaving the modes_pointer */
                               /*                 uninitialized */
}
```

All because of a `switch` statement…

- The programmer wanted to break out of the `if` statement but
  - `break` gets you out of the nearest *enclosing iteration or switch statement.*
  - In this code it broke out of the switch, and executed the call to `use_modes_pointer()` but the necessary initialization had not been done, causing a failure further on.
- This code eventually caused the first major network problem in AT&T's 114-year history.
- The supposedly fail-safe design of the network signalling system actually spread the fault in a chain reaction, bringing down the entire long distance network…and it all rested on a C switch statement!

Making code more understandable…

Handy Heuristic

Making String Comparison Look More ***Natural***

The Problem with `strcmp()`

- One of the problems with the `strcmp()` routine to compare two strings is that it returns **zero** if the strings are identical.
- This leads to convoluted code when the comparison is part of a conditional statement:

  if (!strcmp (s, ”volatile”)) return QUALIFIER;

- A zero result indicates **false**, so we have to *negate* it to get what we want.

Re-Define `strcmp()`

- Use a definition so that the code expresses what is happening in a more natural style.
- Set up the definition:

  #define STRCMP(a,R,b) (strcmp(a,b) R 0)

- Now you can write a string in the natural style

  if (STRCMP (s, ==, “volatile”)) …

Can we do better than the Deep C?

```
int strequal ( char *stringA, char *stringB );
```

- This function returns 1 if the strings are the same and 0 when they are not so that you can put the following in your code:

```
if ( strequal ( argv[1], argv[2] ) ) {
        printf ( "Equal\n" );
} else {
        printf ( "Not equal\n" );
}
```

Just a little confusing…

Overloading

Why use two symbols when one will do?

Overloading *

```
p = N * sizeof * q;
```

- Quickly now, are there **two** multiplications or only **one**?
- The answer is that there's only **one** multiplication.
- `sizeof` is an operator that takes as its operand the thing pointed to by `q`, in other words `*q`.
- When `sizeof` 's operand is a **type** it has to be enclosed in parentheses, but for a **variable** this is not required.

A little more complicated…

- `apple = sizeof (int) * p;`
- What does this mean?
  - Is it the size of an int, multiplied by p?
  - Or the size of whatever p points at cast to an int?
  - ??

Space…the final frontier

No Space – Take a Guess

?? What The ???

Spaces Do Make a Difference

- What do you think the following code means?

```
z = y+++x;
```

- Does it mean?

```
z = y + ++x;
```

```
z = y++ + x;
```

Maximal Munch Strategy

- The ANSI standard specifies a convention that has come to be known as the maximal munch strategy.
- Maximal munch says that if there's more than one possibility for the next token, the compiler will prefer to bite off the one involving the longest sequence of characters.
- So `z=y+++x` will be parsed as `z = y++ + x`.

Munch,Munch, Munch

- But what about

```
z = y+++++x;
```

- Maximum munch will generate:

```
z = y++ ++ + x;
```

```
noSpaces.c:27:11: error: expression is not
assignable
   z = y+++++x;
       ~~~^
```

- **But this is an error!**

- Is there a valid interpretation?

```
z = y++ + ++x;
```