21. Paul Edwards, *The Closed World: Computers and the Politics of Discourse in Cold War America*.

22. Marcel Proust, *Remembrance of Things Past. Volume 1: Swann's Way: Within a Budding Grove*, translated by C. K. Scott Moncrieff and Terence Kilmartin, 50.

■

Obfuscated Code[1]

Nick Montfort

Although conventional wisdom holds that computer programs must be elegant and clear in order to be admirable, there are unusual counterexamples to this principle. In the practice of obfuscated programming, the most pleasing programs are held to be those that are concise but which are also dense and indecipherable, programs that run in some sort of surprising way.[2] Obfuscated code demonstrates that there are other aesthetic principles at play besides those "classical" ones that have been most prominent in discussions of programming aesthetics by programmers[3] and critics.[4]

A popular form of programming related to obfuscation was already in evidence by the beginning of the 1980s. This was the practice of writing one-line BASIC programs, undertaken by people who, for the most part, were not professional programmers, but who had started programming during the home computing boom. These recreational one-liners work in some amusing way, sometimes even implementing a simple interactive game. The following program, for instance, when run on a Commodore 64, displays random mazes:

```
10 PRINT CHR$(109+RND(1)*2); : GOTO 10
```

This is accomplished by simply printing one of two graphic characters at random, "\" or "/", and then, without printing a linebreak, jumping back to the start of the line. The idea of the one-liner is not original to the home computer era and BASIC; in a 1974 talk, Donald Knuth pointed out a precedent in APL programming and noted he enjoyed writing programs that fit on a single punched card.[5] But the one-liner became widespread as BASIC gained popularity, and some one-line BASIC programs (on systems that permit lines longer than eighty characters) became quite intricate and elaborate. A small but reasonably complete implementation of *Tetris* was done in one line of BBC

Micro BASIC in 1992,[6] and a one-line BASIC spreadsheet program has been posted on Usenet.[7] One-line BASIC programs were often printed in magazines and keyed in by users. Code compression, rather than obfuscation for its own sake, was emphasized, but presentations of these programs sometimes asked the reader to figure out what they did, indicating that these programs were puzzling and challenging to decipher.

This puzzle aspect highlights the two main "readers" for a computer program: on the one hand, the human reader who examines the code to understand how it works, and how to debug, improve, or expand it; on the other, the computer, which executes its statements or evaluates its functions by running the corresponding machine code on its processor. A program may be clear enough to a human reader but may have a bug in it that causes it not to run, or a program may work perfectly well but be hard to understand. Writers of obfuscated code strive to achieve the latter, crafting programs so that the gap between human meaning and program semantics gives aesthetic pleasure.

Obfuscated programming is institutionalized today not in microcomputer magazines but online, where programs are exchanged and contests are hosted. The International Obfuscated C Code Contest has been held eighteen times since the first contest ran in 1984, back when one-line BASIC programs were still in vogue. Only small, complete C programs can be entered in the IOCCC. The contest's stated goals include demonstrating the importance of programming style "in an ironic way" and illustrating "some of the subtleties of the C language."[8] There is also an obfuscated Perl contest, run annually by *The Perl Journal* since 1996, but the most visible tradition of Perl obfuscation is seen in short programs that print "Just another Perl hacker," which are called JAPHs. In early 1990, Randal Schwartz began the tradition of writing these programs by including them in his signature when posting on comp.lang.perl.

Some sorts of obfuscation techniques are common to IOCCC entries and JAPHs and may be used in just about any programming language. Even assembly language allows the free naming of variables and labeling of particular instructions, so that these names can be used meaningfully and can help people better understand programs. Wherever such names can be freely chosen, they can be selected in a meaningless or even a deceptive way, as when num or count is used to store something other than a number, or when x and y appear together in a program to mislead the reader into thinking they are Cartesian coordinates. Since variable names are usually case-sensitive, there are addi-

tional possibilities for confusion. In C, where no special character is used to indicate a variable, programs take advantage of this and of the case-sensitivity of variable names to name some variables o and O, inviting additional confusion with the number zero. This play, which can be called *naming obfuscation*, shows one very wide range of choices that programmers have. By calling attention to this, naming obfuscation demonstrates that everything about a programmer's task is not automatic, value-neutral, and disconnected from the meanings of words in the world.

Another obfuscation technique takes advantage of curiosities in syntax to make it seem that some piece of data—for instance, a string that is being assigned to a variable—is actually part of the program's code. Alternatively, something that appears to be a comment, and thus to have no effect on the program's workings, may actually be part of the code, or vice-versa. This *data/code/comment confusion* is invited by flaws or curiosities in a language's specification, but can be accomplished in several different languages, including C and Perl.

There are also obfuscations that appear more prominently in one language than in another. In C, a[b] and b[a] have the same meaning, which is not the case when accessing array elements in other languages. An obfuscator working in C, however, can choose the more confusing of the two. Other languages do not define the addition of strings and numbers, or they define it in a way that seems more intuitive, at least to beginning programmers. But C, by giving the programmer the power to use pointers into memory as numbers and to perform arithmetic with them, particularly enables *pointer confusion*. By showing how much room there is to program in perplexing ways—and yet accomplishing astounding results at the same time—obfuscated C programs comment on particular aspects of that language, especially its flexible and dangerous facilities for pointer arithmetic.

Perl does not invite this sort of obfuscation, but does allow for several others. There are a dazzling variety of extremely useful special variables in Perl, which are represented with pairs of punctuation marks; this feature of the language merits an obfuscation category of its own. Perl's powerful pattern-matching abilities also allow for cryptic and deft string manipulations. The name Perl is sometimes said to stand for "Practical Extraction and Report Language," but "Pathologically Eclectic Rubbish Lister" is sometimes mentioned as another possible expansion. The language is ideal for text processing, which means that short messages (such as "Just another Perl hacker,") can be printed out in

many interesting ways, sometimes using little-known sorts of *pattern-matching obfuscation*.

This JAPH, posted by Randal Schwartz on April 18, 1990, provides a short example that can be explicated in some depth:

```
$_=",rekcah lreP rehtona tsuJ";s/.$/eval 'print $&',""/e while
length
```

Like most such programs, this one prints "Just another Perl hacker,"—the comma at the end is traditional—and does so in a curious way. There are only two statements in this one-line program, separated by a semicolon. The first statement puts a string with the reverse of this message into $_, the Perl special variable for the current line. The second command is the interesting one; it is a substitution operation of the form s/FIND/REPLACE/e which is called implicitly on $_. The e after the final slash means that the result will be evaluated as a Perl expression. The "while length" at the very end results in this substitution being repeatedly called, iteratively, as long as there is something left in $_. Since one character is removed from the string on each pass, the following substitution operation is called once for each character in the string:

```
s/.$/eval 'print $&',""/e
```

The effect of this is to take the last character in the current line—"J" will therefore be selected first—and prepare a string to contain it. The first such string that is built is "eval 'print_,"""". This string is evaluated as a Perl expression, which results in "eval" executing its own Perl program to print the character "J". Since this mini-program returns no value, the letter selected is replaced with nothing, and the string is diminished in length.

There would be nothing very interesting about simply reversing a string and then printing it out, or about starting at the end of a string and printing it back-to-front one character at a time, although it might be interesting to see one of these processes coded up in a single, short statement. Here, a single statement does all of this and more. The statement creates a string that, when evaluated as an expression, executes a very short program to print a character. This statement also removes that last character from the current line and then continues processing the shorter line.

A repository of JAPHs is available online[9] and explications of several have been provided.[10] An explication of an introductory obfuscated C program[11] is also available.

Recent IOCCC programs include a racing game in the style of Pole Position, a CGI-enabled web server, and a program to display mazes whose code is itself in the shape of a maze. Obfuscated code in Perl as well as C often spells out a name in large letters or assumes the form of some other ASCII art picture. This is a type of *double coding*; more generally, *multiple coding* can be seen in "bilingual" programs, which are valid computer programs in two different programming languages. Double coding in natural languages is exemplified by the sentence "Jean put dire comment on tape," which is grammatical English and grammatical French ("Jean [male name] is able to say how one types"), although each word has a different meaning in each language. Harry Mathews contributed to further French/English double coding by assembling the Mathews Corpus, a list of words which exist in both languages but have different meanings.[12] In programming, an important first step was the 1968 *Algol* by Noël Arnaud, a book of poems composed from keywords in the Algol programming language.[13] Perl poetry is a prominent modern-day form of double-coding, distinguished from obfuscated programming as a practice mainly because it is not as important in Perl poetry that the program function in an interesting way; the essential requirement is that the poem be valid Perl.

Interestingly, it is not the case that languages typically despised by hackers—for instance, COBOL and Visual Basic—are the main ones used in obfuscation. Many Perl hackers and C coders who write obfuscated programs also use those languages professionally and find it enjoyable to code in those languages. They generally do not find it fun to program in COBOL or Visual Basic, however, even to comment negatively on these languages. In addition to making fun of some "misfeatures" or at least abusable features of languages, obfuscated code shows how powerful, flexible programming languages allow for creative coding, not only in terms of the output but in terms of the legibility and appearance of the source code.

All obfuscations—including naming obfuscations as well as language-specific ones, such as choosing the least well-known language construct to accomplish something—explore the *play* in programming, the free space that is available to programmers. If something can only be done one way, it cannot be obfuscated. It is this play that can be exploited to make the program signify on different levels in unusual ways.

The practice of obfuscated programming, like the kindred practice of developing weird programming languages, is connected to certain literary and artistic traditions. The practice suggests that coding can resist clarity and elegance to strive instead for complexity, can make the familiar unfamiliar, and can wrestle with the language in which it is written, just as much contemporary literature does. Another heritage is the tradition of overcomplicated machinery that has manifested itself in art in several ways. Alfred Jarry's 'Pataphysics, "the science of imaginary solutions," which involves the design of complicated physical machinery and also the obfuscation of information and standards, is one predecessor for obfuscated programming. There are also the kinetic installations of Peter Fischli and David Weiss and the elaborate apparatus seen in their film *The Way Things Go* (1987–1988), as well as the earlier visual art of Robert Storm Petersen, Heath Robinson, and Rube Goldberg. These depictions and realizations of mechanical ecstasy comment on engineering practice and physical possibility, much as obfuscated coding and weird languages comment on programming and computation. Such "art machines" anticipate obfuscated programs by doing something in a very complex way, but also by actually doing *something* and causing a machine to work.

Obfuscated code is intentionally difficult to understand, but the practice of obfuscated programming does not oppose the human understanding of code. It darkens the usually "clear box" of source code into something that is difficult to trace through and puzzle out, but by doing this, it makes code more enticing, inviting the attention and close reading of programmers. There is enjoyment in figuring out what an obfuscated program does that would not be found in longer, perfectly clear code that does the same thing. While obfuscation shows that clarity in programming is not the only possible virtue, it also shows, quite strikingly, that programs both cause computers to function and are texts interpreted by human readers. In this way it throws light on the nature of all source code, which is human-read and machine-interpreted, and can remind critics to look for different dimensions of meaning and multiple codings in all sorts of programs.

Notes

1. Parts of this article are based on a paper entitled "A Box Darkly" that Michael Mateas and I presented at Digital Arts and Culture 2005.

2. There is also a practice of making one's code difficult to understand or reverse-engineer for commercial purposes, to keep competitors or clients from understanding one's proprietary programs. Despite some similarities in what is done in this case, this practice seems to have no aesthetic principle behind it and no important relationship to obfuscated programming as described here.

3. For example, Donald E. Knuth, "Computer Programming as an Art," in *Literate Programming*, 1–16.

4. For example, Maurice J. Black, "The Art of Code," Ph.D. Dissertation, University of Pennsylvania (2002).

5. Knuth, "Computer Programming as an Art."

6. David Moore, "Rheolism: One Line Tetromino Game," available at http://www.survex.com/~olly/dsm_rheolism/ (accessed July 1, 2001).

7. Mark Owen, "BASIC Spreadsheet." Quoted in C. D. Wright, "One Line Spreadsheet in BASIC," post to comp.lang.functional. Message-ID: <D01s7J.LK3@cix.compulink.co.uk> (November 29, 1994).

8. Landon Curt Noll, Simon Cooper, Peter Seebach, and Leonid A. Broukhis. "The International Obfuscated C Code Contest," available at http://www.ioccc.org/main.html/.

9. JAPHs, available at http://www.cpan.org/misc/japh/.

10. See Teodor Zlatanov, "Cultured Perl: The Elegance of JAPH"; Abigail, "JAPHs and Other Obscure Signatures"; and, Mark-Jason Dominus, "Explanation of japh.pl"

11. Michael Mateas and Nick Montfort, "A Box Darkly: Obfuscation, Weird Languages and Code Aesthetics."

12. Harry Mathews and Alistair Brotchie, *Oulipo Compendium.*

13. Ibid.