ON THE TYPESETTING OF COMPUTER PROGRAMS

ESMOND PITT

The following remarks are offered to designers and typesetters who have to set texts containing computer program text, and for the information of authors of such texts. I write from the twin backgrounds of 29 years's experience in computer programming and a lifelong interest in typography.

There are sound reasons, some good, some bad, for the problems involved in typesetting computer programs. In many cases, these problems have simple solutions which will not be found abhorrent either to typographers or programmers, but which may not be obvious to either group.

> The terms 'code', 'computer code', 'computer programs', 'program text', and 'program code' all appear in these notes, always with the same meaning.

1 Program code is a scientific notation

Typographers are primarily concerned with legibility, and have tools, practices, and traditions dating back hundreds and indeed thousands of years on which to rely when setting texts in natural languages. However, computer programs are not written in natural languages. They are written in 'programming languages': artificial languages, which have their own rules of syntax, their own conventions of presentation, and their own criteria of legibility. Computer code is therefore a special domain for typesetting, just as are music, mathematics, and chemistry. These domains have their own rules, which are not the rules used when setting natural languages.

As recently as 1978 the prominent computer scientist Donald Knuth became so dissatisfied by existing techniques for typesetting of mathematics that he invented the $\TeX$ system for computer typography. The typesetting of mathematics is hundreds of years older than the typesetting of computer programs. Computer programming itself is of very recent origin, and the practice of setting it in type doesn't go back more than about 45 years: significant volumes of

computer code have only been published in the last 20 years or less. The associated typographical discipline is immature or indeed practically non-existent, and the typographical expectations of the practitioners in the field are also low.

There are differing ideas within the profession about what constitutes good presentation of computer programs in their original medium, and there is no reason to consider this odd.[1] There are many programming languages, although they can be grouped into families, each broadly associated with a visual style.[2] Sometimes the visual style is more or less compulsory (i.e. enforced by the computer). More usually, visual style is conventional: the result of a very broad agreement, subject to a surprisingly small set of variations. Recent 'intelligent' program editors can be configured to observe a number of common, independent variations in programming style: this servers to demonstrate that although overall style can be mechanised, programmers, including author-programmers, tend to each have their own recognizable stylistic quirks.

2 Computers are typographically impoverished

The 'look and feel' of computer input and output derives not from millennia of scribal lore and centuries of typographic practice, but from mechanical tabulation and accounting systems, 80-character Hollerith punched cards, telex machines, Teletype machines, 132-column uppercase-only line printers, dot-matrix terminals and printers, and 24 × 80 visual display units (VDUs). The long and distinguished history of typography did not impinge on most of these developments.

The character set itself is impoverished. The reasons for this were primarily economic. Early computers were extremely expensive, and were designed for mathematical and engineering convenience and economy, aesthetic concerns being secondary (or non-existent). Computer characters came from minimalist character sets like EBCDIC and ASCII, designed to fit into a byte.[3] In early mainframes a byte was 6 bits, capable of holding 64 EBCDIC values; later, the 8-bit byte, capable of holding 256 ASCII values, became universal. By contrast, Gutenberg's compositing case for the 42-line Bible already held 290 'sorts'.[4]

Computer languages themselves do not derive from English,[5] French, or other natural languages, but from artificial languages like algebra & set theory from mathematics, and propositional & predicate calculus from formal logic.

1. Compare the free use of line breaks and indentation within any mathematical text.
2. This is comparable to the existence of both Newton and Leibniz notations for calculus.
3. Unicode provides for at least 2^{16} characters; however it is a code for *characters*, not for compositors' sorts, lacking e.g. ligatures, non-ranging numberals, small capitals, compositor's dashes and spaces, &c.
4. Bringhurst, *The Elements of Typographical Style*, p. 180.
5. With the notable exception of COBOL.

3 Typography of programming languages

For typographic purposes, programming languages can be divided into two groups:

(a) 'Columnar' languages: these are the early languages such as FORTRAN, COBOL, RPG, in which the language is defined on an 80-character columnar format (80 columns of one character each): certain columns, or adjacent groups of columns, have specific purposes. Modern 'assembly' languages continue to use columnar formats.

(b) 'Free-form' languages: in these languages, all white space, including line and page breaks, is insignificant except insofar as it separates word from word (i.e. is treated by the computer as a single space). This group includes most programming languages designed since the early 1960s, certainly including C, C++, Java, JavaScript, the varieties of Basic, Pascal, SQL, the varieties of Algol, PL/1, ….[1]

The columnar languages came first, as being easier for humans to conceive, define, and comprehend. The free-form languages came about on the realization that it is easier for the computer to completely ignore all two-dimensional aspects of the program and just view it as a linear sequence of characters. However, this is not easier for the reader. Practically all programmers use highly disciplined two-dimensional layout techniques, which remain easier for humans to conceive and comprehend.

> It is possible to format any free-form language program as a more or less left- and right-justified paragraph of continuous text, but the result is entirely incomprehensible by a human. This technique is often used by 'source code obfuscators', to conceal the workings of a program while making the source code publicly available: this is the clearest possible evidence that natural language typography cannot be sensibly applied to computer programs, precisely because the result is illegible.

Programming languages, whether columnar or free-form, exhibit common typographic traits, which mostly derive from the hardware originally used as computer input and output devices as mentioned above. These are discussed in the subsections following.

3.1 Indentation is part of the notation

The indentation from the left-hand margin is critical to the legibility of program code. In columnar languages, indentation from the left affects how the text is understood by the computer, i.e. is part of the syntax of the language. In free-form languages, indentation no longer conveys meaning to the computer, but it still conveys critical meaning about logical structure to the reader.[2]

In this respect, indentation in computer programs is no different from indentation in natural languages: consider multi-level lists, which represent a logical structure. Deeply

1. The free-form languages can themselves be further divided in ways which are not relevant to this essay.

nested logical structures are endemic to computer programming, and cannot be merely avoided or recommended against, as in natural languages. A typographer's function is to convey the author's intention legibly: in computer programs a great deal of the intent is represented by indentation.

3.2 Line breaks are part of the notation

Line breaks in computer code are subject to the rules of the language, not the rules of English or typography. In practically all programming languages, line breaks are elements of the syntax. This is trivially true in the case of columnar languages, but it is also true in most free-form languages:[1]

(a) in many of these languages (e.g. forms of Basic), line breaks terminate statements

(b) special syntax is generally required to continue quoted strings accross line breaks

(c) most of these languages have comment formats in which the line break terminates the comment.

3.3 Quotation marks are part of the notation

Most programming languages use double quotes to delimit strings (sequences of one or more characters in the computer), and single quotes to delimit single characters (or, confusingly for the typographer, program notations which represent single characters in the computer, e.g. '0x0a' or '\n').

These are invariably rules of the programming language, against which typographical practice—specifically, the typographic preference for single quotes—cannot prevail.

Usually the author's text will have been checked for errors by the computer (the 'compiler'); in any case the typographer's function must be to set it as submitted.

3.4 Set width

The expected set width of computer program text is 80 characters. Typographically speaking, this is not actually excessive, but it is well above the norm of 60–65 characters for a book page. Historically, the 80–character width arose because programs were first notated on Hollerith cards,[2] which were already ubiquitous in automatic tabulation systems. For backward-compatibility reasons, the same width had to be supported by subse-

2. In free-form languages, meaning about logical structure is represented to the computer by { and }, or 'begin' and 'end'. Although the computer doesn't care how these relate visually, it is critically important for the reader to be able to quickly relate a { to its closing }, which may be many lines away. It is precisely *because* the computer doesn't care that the visual alignment of these is so critical.

1. Certainly this is true of the mainstream free-format languages C, C++, Java, JavaScript, and Visual Basic. In fact it is difficult to identify a free-format language in which one or other or these principles does not apply: possibly the original Pascal or some of the micro-versions of PL/1.

quent equipment used for entering programs: Teletype terminals, dot-matrix terminals, visual display units, and, ultimately, graphical program editors.

Fortunately there is a way to reduce this set width without using tiny fonts: see §4.

3.5 Justification

Computer programs are always written, viewed, and set left-justified, right-ragged. This is a matter of legibility, the typographer's primary concern. As noted above, paragraphs of code set solid and justified left and right are utterly illegible. In most cases they are also incorrect by virtue of §3.2.[1]

3.6 Sans serif

Computer programs are conventionally displayed in sans-serif fonts. This is largely for the historical reasons already given.

It is also sound typographic practice, as it sets off program code from the narrative very clearly, especially when words of program text have to appear within the narrative.

Solid paragraphs of program text do not occur, and program code is scrutinized rather than speed-read, so the reduced legibility of sans-serif fonts when used *en masse* is not an issue.

3.7 Monospacing

Historically, all the equipment used for entering computer programs used monospaced fonts, from Hollerith cards, line printers, Teletype machines, dot-matrix terminals, and so on to VDUs and early graphical editors.

In columnar programming languages, certain elements *must* appear in certain columns or groups of columns. For this reason, use of a monospaced font is essential in those languages.

The practice of monospacing has carried over into the later free-form languages by virtue of hardware, history, and habit. Even when graphical editors became available, the historical 'pull' of this legacy is so strong that, by default, monospaced fonts continue to be used for program text. I don't accept that this is necessary: see my recommendations in §4.

2. And printed on 132-column line printers in upper-case only. It could have been worse: Remington Rand sold punch cards with 90 and even 130 columns in the era of office tabulation. Fortunately for typographers, they only had 15% or so of the market.
1. If further discussion were needed, it might be pointed out that programs are not written in paragraphs at all; they are composed of nested logical 'blocks'.

3.8 Tab stops

Tab stops of 8 spaces are commonly employed in program code. This was originally a feature of the hardware discussed above. It relates strongly to the generous 80–character page width which was available.

This 8–space standard for tab stops is obviously far too wide. When programmable tab widths became available some 20 years ago on hardware, I reduced my own tab stops to 4 spaces, and more latterly, when graphical editing software became available, to 2. See also my recommendations in §4.

3.9 Page breaks

When setting computer code in a book, page breaks can't just follow the simple orphan/widow principles used when typesetting natural languages. Instead, the logical 'blocks' of the code must be kept together if possible. It is not usually possible for the typographer to determine the block boundaries in code, although a blank line is generally an acceptable point for a page break. 'Block comments' should be kept with the following block of code. If you don't know what these are, ask the author.

3.10 Hyphenation

Programming languages are not natural languages and do not observe the usual hyphenation conventions. Instead:

(a) Columnar languages use a 'continuation character' in a specific column to indicate either (a) that this line continues the previous line, or (b) that this line is continued by the next line, depending on the particular language.

(b) Free-form programming languages are designed such that all white space is insignificant, and it is up to the programmer to break the line at a visually or logically sensible place.

Neither of these practices can be reduced to typographic rules for hyphenation. Instead, the MS must be set as submitted, or the author must be consulted.

3.11 Program words

Mathematicians generally use single letters for variables, and symbols for operators. Programming languages allow programmer-defined words—up to 30 characters or more—for variables, and the built-in operators of the language often appear as words as well.

Programmers freely form compound words: either:

(a) along Germanic lines: e.g. PendingEventQueue, ClientSocket, ByteArrayInputStream, or

(b) by pseudo-hyphenation with the underscore character: e.g. BUFFER_SIZE, MAX_THREADS, &c.

In 'object-oriented' languages, words can further be compounded with punctuation, according to language-specific rules, e.g. String::length (C++), or java.lang.String (Java). The latter is a 'fully-qualified Java class name': these are often problematic when typeset in the narrative, as they frequently run over desired or necessary line-breaks. As with URLs, frequently the only answer is to set them in a separate line.

Words in program text must *never* be hyphenated or line-broken except in accordance with the author's instructions.

3.12 Upper and lower case

Case in program code is usually significant to the computer, and practically always to writers and their readers.

Pairs of words are often used which differ only in case, representing different things: e.g. BufferedOutputStream and bufferedOutputStream.

Programmers, especially author-programmers, are usually highly systematic about case, in ways which may not necessarily make sense to the typographer (or other programmers!).

4 Practical recommendations

The following practices are recommended when typesetting computer programs.

4.1 Indent in em units

The solution to many of the issues in typesetting computer programs is the em. The author's tabs will most likely be to the next multiple of 8 spaces (1, 9, 17, ...); typographic tabs for program code should be in multiples of 1 or 2 ems.

Adopting the em as the unit of indentation may at first 'look funny' to the author, as the indents may be much narrower than seen on screens or printouts. However, as long as the vertical alignment of tab stops is preserved, the author's intention is fully preserved.

It is not necessary to preserve *all* vertical alignment, only the vertical alignment at tab stops. The author may need to be convinced of this, but it is true.[1]

1. Difficulties can arise if the author doesn't use tabs correctly, by using multiple spaces as well as tab stops, e.g. combinations of 4 spaces and 8 -space tab stops. This is poor practice: most editing software can be configured to (a) convert all n-space sequences to tabs and (b) display all tabs as n spaces, which is what should be done. In the source file, tabs should be tabs, and their visual representation should be chosen by the viewer. This is also the justification for setting tabs as ems instead of 8 spaces. Difficulties can also arise if the author uses tabs in the middle of lines, i.e. after left-indentation tabs and other text.

Setting tabs in em units has the following additional beneficial effects:

(a) It is good typography, for the same reasons as in natural languages: it obeys the horizontal rhythm of the line.

(b) It reduces the appetite for the 80-character line width, allowing a more moderate set width to be used, or a larger font.

(c) Similarly, it reduces the appetite for hyphenation and line breaks. In fact it may allow lines which the author had to break to be set unbroken: the typesetter may wish to consult with the author after doing a proof, to refine the setting of the program code.

(d) In free-form languages, it eliminates the need for monospaced fonts. As the em is a constant width, the logical structure of the program is still clearly expressed, without relying on all characters being a constant width.

4.2 Fonts

If unsure, ask the author whether the language is free-form or columnar.

Columnar languages should always be set in a monospaced font.

Free-form language programmers are used to seeing Courier on their terminals and screens, and Courier or Helvetica in print. This applies even to programmers who have never seen a columnar language. This is typographically unfortunate: worse, there is a strong prejudice in favour of monospaced fonts even for free-form languages. As I have shown above, this is unnecessary: proportional fonts can certainly be used as long as vertical alignment of tab-stops is preserved.

Recent graphical editors use a mix of fonts for displaying programs. As graphical editors are becoming more widely used, it seems to me quite legitimate to imitate in print what they do on the screen. Typically this is as follows:

(a) program text: a sans-serif font (not necessarily monospaced)

(b) comments: italic face of the same font, or sometimes italics of a Roman font

(c) 'keywords' of the language ('if', 'while', etc.): boldface of the program text font.[1]

One designer recommends TheSansMonoCondensed, by Lucas de Groot. As mentioned above, as a programmer I can see no objection to proportionally-spaced fonts for free-form languages, as long as vertical alignment at tab stops is preserved.

I would also give serious consideration to using faces in which the roman and sans-serif are strongly related, e.g. Scala and Scala Sans, or Quadraat and Quadraat Sans, using the roman form for the narrative.

1. This is definitely an optional 'frill' , requiring the author's detailed assistance or markup.

Prejudices do exist against proportional-spaced fonts for computer code, or indeed against anything but Courier, but these are declining as graphical editors become more widely used, and as better books are published.

See also §5.

4.3 Line breaks

Line breaks must be as per MS. If a line break must be set where the MS has none, consult the author, who may specify a different break point, or may rewrite the affected code. Usually the follow-on must be indented *at least* to the current indentation level, but again the author is the only possible guide.

If the suggestion at §4.1 is followed, this situation will rarely arise; indeed, the opposite situation will arise—i.e. being able to set one line where the MS has a broken line. Such opportunities must be discussed with the author, or obey MS markings.

Consult the author for guidelines about line-breaks in code words when set in the narrative. The guideline may well be 'don't'!

4.4 Page breaks

If page breaks may occur in the middle of program code, the author must be consulted as to preferred page break points. Usually this is to be avoided altogether in short examples; in longer programs, the author should indicate all possible page breaks in the MS.[1]

Have the author divide code paragraphs into moderate-sized paragraphs, and use a 'keep all lines together' setting if available; otherwise, use a widow/orphan setting larger than or equal to the author's maximum code paragraph size.

See also §3.10.

4.5 Quotes

Conventionally, 'straight' quotes are used, not typographic quotes. This is historically determined, by the use of fonts without typographic quotes (e.g. Courier, Helvetica) in typeset computer code. It is not required by the properties of the notation.

I see no reason against using typographic quotes when setting computer programs as long as single quotes stay single and double quotes stay double, i.e. as long as the author's quotes are preserved rather than 'corrected' to standard typographic practice.

4.6 Numerals

Conventionally, lining numerals have always been used in program code, for the historico-hardware reasons already given. If you can be bothered using old-style numerals in

1. In publishing books, it is fair to wonder whether long programs should be printed at all. They are more usful in electronic form, i.e. on a CD or Web page—unless there are cross-references to the code from the text, in which case the author and deisgner need to consider whether line numbers should be shown.

program code, or if the font is built that way, I can see no reason against it. It is best to use a font in which 1, I, and l (lower-case L) are distinct, as also 0 (zero) and O.

5 Examples

The following examples show how a program text may appear: in the MS as written by the author, and when typeset in various different ways.

The example is written in the Java language. It will be seen that although the typeset examples are much more compact horizontally, all essential vertical alignment is preserved, even in the proportional font.[1]

Where possible, I have set comments in italic.

These examples can be used to convince sceptics. Sometimes the strongest argument for tabbing to ems is that it conserves the width when, as usual, you don't have the author's original 80–character width available.

5.1 Code as per MS

10 pt Courier, tab stops every 8 spaces.

```
public class Queue extends Vector
{
        public synchronized void enqueue(Object object)
        {
                super.add(object);
                notifyAll();     // waken waiters
        }

        public synchronized Object dequeue()
        {
                while (size() < 1)
                        wait();
                return super.remove(0);
        }
}
```

5.2 Code as per MS, alternate format

10 pt Courier, tab stops every 8 spaces. This example demonstrates another possible format the author might have used. Note that in this format the opening brace { is placed on the end of the line, and the closing brace } aligns with the beginning of the line which contains the matching opening brace. Many programmers find this layout preferable. I

1. i.e. (in this language) the vertical alignment of matching pairs of { and }, and the consecutive indentations they enclose.

don't. This format does have the typographic advantage of economizing slightly on vertical space.

```
public class Queue extends Vector {
        public synchronized void enqueue(Object object) {
                super.add(object);
                notifyAll();     // waken waiters
        }

        public synchronized Object dequeue() {
                while (size() < 1)
                        wait();
                return super.remove(0);
        }
}
```

5.3 Code typeset in a monospaced font

9.13 pt Lucida Sans Typewriter, laterally condensed to 95% of the resulting size, tab stops every em.[1]

```
public class Queue extends Vector
{
  public void enqueue(Object object)
  {
    super.add(object);
    notifyAll();// waken waiters
  }

  public synchronized Object dequeue()
  {
    while (size() < 1)
      wait();
    return super.remove(0);
  }
}
```

5.4 Code typeset in a proportional font

11 pt Gill Sans, tab stops every em: not necessarily as a recommendation, but just to demonstrate that vertical alignment can be preserved even with a proportional font.

1. This set at 83 % of point size to match narrative text set in a roman font.

```
public class Queue extends Vector
{
  public void enqueue(Object object)
  {
    super.add(object);
    notifyAll();// waken waiters
  }

  public synchronized Object dequeue()
  {
    while (size() < 1)
      wait();
    return super.remove(0);
  }
}
```

5.5 Optima (Hermann Zapf)

I think this is a rather good choice, as it avoids the bare appearance of most sans-serifs while still preserving a difference from serif fonts. It will be seen that it goes quite nicely with this copy, which is Aldus body text and Palatino headings.

```
public class Queue extends Vector
{
  public void enqueue(Object object)
  {
    super.add(object);
    notifyAll();  // waken waiters
  }

  public synchronized Object dequeue()
  {
    while (size() < 1)
      wait();
    return super.remove(0);
  }
}
```

5.6 Code typeset in a roman font

11 pt Garamond, tab stops every em: again, not necessarily as a recommendation, but simply to demonstrate to sceptics that vertical alignment in a free-form (non-columnar) language like Java can be preserved even with a proportional roman font.

```
public class Queue extends Vector
{
  public void enqueue(Object object)
  {
    super.add(object);
    notifyAll();  // waken waiters
  }

  public synchronized Object dequeue()
  {
    while (size() < 1)
      wait();
    return super.remove(0);
  }
}
```

5.7 Code typeset in a roman font, justified

11 pt Garamond, tab stops every em: again. This ridiculous example is provided simply to demonstrate why the author's linebreaks and left-indentation are aspects of legibility. As a matter of fact this rendition is not even correct, because the comment 'waken waiters' is syntactically terminated by the following linebreak, which is elided here.

```
public class Queue extends Vector { public void enqueue(Object object) { super.add(object);
notifyAll(); // waken waiters } public synchronized Object dequeue() { while (size() < 1)
wait(); return super.remove(0); } }
```

6 Author

The author is a computer consultant and writer, and typesets his own books.

Address: Melbourne Software Company, 286 Canterbury Rd, St Kilda West, Victoria 3182, Australia; esmond.pitt@bigpond.com.

7 Colophon

This text is set in 11/13 pt Aldus, using AldusSC small caps and numerals for body text and footnotes, and Palatino and PalatinoSC in the titles and headings. The examples are variously set in Courier, Lucida Sans Typewriter, Gill Sans, and Garamond. The e-mail address is set in Hermann Zapf's Optima. Copy was prepared with Adobe FrameMaker 7.0p576 running on Windows 98se, and distilled to pdf with Acrobat Distiller 5.05.