Design Principles for the Enhanced Presentation of Computer Program Source Text

Ronald Baecker
Dynamic Graphics Project
Computer Systems Research Institute and Department of Computer Science
University of Toronto
Toronto Ontario M5S 1A4 Canada

Aaron Marcus
Aaron Marcus and Associates
1196 Euclid Avenue
Berkeley California 94708

Abstract

In order to make computer programs more readable, understandable, appealing, memorable, and maintainable, the presentation of program source text needs to be enhanced over its conventional treatment. Towards this end, we present five basic design principles for enhanced program visualization and a framework for applying these principles to particular programming languages. The framework deals comprehensively with ten fundamental areas that are central to the structure of programming languages. We then use the principles and the framework to develop a novel design for the effective presentation of source text in the C programming language.

Keywords

User interface design, human factors, graphic design, program visualization, prettyprinting, program beautification, computer typesetting, computer program documentation, software engineering.

Introduction

In two earlier papers [Marcus & Baecker 82, Baecker & Marcus 83], the authors suggested a new approach to the presentation of computer program source text, presented two ad hoc examples illustrating the potential of the idea, and outlined a research strategy with which to develop the approach systematically.

The approach is to employ graphic design and high resolution bit-mapped displays and laser printers to produce far richer representations of program source than those employed conventionally. This can be done by using multiple fonts, variable point sizes, variable character widths, proportional character spacing, variable word spacing and line spacing, non-alphanumeric symbols, gray scale tints, rules, and arbitrary spatial location of elements on a page.

The goal of the work is to enhance the interface to a computer program's source text, and thereby to make it:

- more readable;
- more understandable;
- more appealing;
- more memorable; and,
- more maintainable.

Our work thus encompasses the field of prettyprinting, an area in which others before us have worked with more limited graphics tools. The earliest work was done on LISP, so that program readers would not drown in a sea of parentheses. The problems of prettyprinting PASCAL have elicited a long correspondence in the ACM SIGPLAN notices [Hueras & Ledgard, 1977; Grogono, 1979, Gustafson, 1979; Leinbaugh, 1980]. A discussion of prettyprinting algorithms and their complexity has appeared [Oppen, 1980]. Other authors [Rose & Welsh, 1977; Rubin, 1983] demonstrated methods of extending the syntactic descriptions of programming languages to include their formatting conventions. One paper [Miara, Musselman, Navarro & Schneiderman, 1983] reviews a number of human factors experiments concerning the effect of program indentation on program comprehensibility. Although their experiment showed that a moderate level of indentation increases program comprehension and user satisfaction, the body of research they surveyed fails to provide clear experimental confirmation of what every programmer knows: a program's appearance dramatically effects its comprehensibility and useability.

Our research [Baecker et. al. 1985] however goes significantly beyond conventional approaches to prettyprinting in five ways:

- The availability of rich typographic and pictorial representations changes the nature of the problem to one that is *qualitatively different* from that of prettyprinting on a line printer.
- We have identified five basic graphic design principles of enhanced program visualization and developed a framework for applying them to particular programming languages.
- We have systematically carried out graphic design experimental variations in order to arrive at a carefully considered set of recommended conventions for the appearance of programs in the C [Harbison & Steele, 1984] programming language. (We chose to work with C because of its commercial importance and its unreadability.)

• We have developed a flexible tool, the SEE *visual compiler*, that automates the production of enhanced C source text. SEE is highly parametric, thus allowing easy experimentation in trying out novel variations, and suiting the variety of style preferences that characterizes the programmer community.

• Finally, we have considered the entire context in which code is presented, a context which includes the supporting text and notations that make a program a *technical publication*, a living piece of written communication.

This paper reports new results which focus on the first three of these contributions by:

• Presenting the five basic graphic design principles for enhanced program visualization and the framework for applying them to particular programming languages. This is done by applying the principles systematically to ten fundamental kinds of programming language constructs.

• Presenting in detail a design of C program appearance based on the design principles, the framework for their application, and the experimental variations that we have carried out.

• Illustrating our design and the application of the design principles with several examples of sample program appearance.

• Briefly describing the experimental tool with which we produced our design variations and some important remaining research problems.

Five Design Principles

For the purposes of enhanced computer program visualization, there are five central organizing principles:

• Principle I – *Typographic Vocabulary*: Choose a small number of appropriate typestyles of suitable legibility, clarity, and distinguishability, applying them thoughtfully to encode and distinguish various kinds of program tokens. Within each typeface, choose or design a set of enhanced letterforms and symbols with which to represent the text effectively.

• Principle II – *Typesetting Parameters*: Adjust to enhance readability the typesetting parameters of text point size, headline size and usage, word spacing, paragraph indentation, and line spacing. Whereas Principle I concerns itself with the selection of symbols, Principle II deals with their application in clusters.

• Principle III – *Page Composition*: Bring out the program structure by carefully structuring the page composition through the use of grids, the application of explicit and implicit lines of varying thickness (typographic "rules"), and the inclusion of appropriate white space.

• Principle IV – *Symbols and Diagrammatic Elements*: Integrate appropriate symbols and diagrammatic and graphic elements to elucidate essential program structure. In this way we achieve non-textual augmentations of the source code proper.

• Principle V – *Metatext*: Augment the source text with mechanically generated supplementary text, additional data and commentaries that enhance the comprehensibility of the source text.

A Framework for Applying the Principles

We now apply this rich palette of graphic design techniques to reveal and express the meaning of ten major categories of programming language constructs:

• Category 1 – *The Presentation of Program Environment.* Clarifying the context in which the program was created, is maintained, and will be used.

• Category 2 – *The Spatial Composition of Comments.* Presenting program comments clearly in relationship to program code.

• Category 3 – *The Typography of Program Punctuation.* Enhancing the visual effectiveness of C punctuation marks (separators, containment symbols, and operators).

• Category 4 – *Typographic Encodings of Token Attributes.* Mapping C tokens (identifiers, reserved words, and constants) into effective typographic representations.

• Category 5 – *The Presentation of Preprocessor Commands.* Presenting C preprocessor commands in a more effective manner.

• Category 6 – *The Presentation of Declarations.* Enhancing the structure of the declarations of C identifiers.

• Category 7 – *The Visual Parsing of Expressions.* Using typographic attributes to enhance the ability of a human reader to identify and understand complex program expressions.

• Category 8 – *The Visual Parsing of Statements.* Using typographic attributes to enhance the ability of the reader to identify and understand complex program statements.

• Category 9 – *The Presentation of Function Definitions.* Clarifying the structure of the definitions of C functions.

• Category 10 – *The Presentation of Program Structure.* Enhancing the structure of a program in terms of its constituent parts, for example, its constituent files, declarations, function definitions, and global variables.

Previous work on prettyprinting has dealt primarily with Category 8 and somewhat with Category 4 (although often badly). The definition of the categories and the exploration of Categories 1, 2, 3, 5, 6, 7, 9, and 10 are original with our research.

A Design for Enhanced C Program Source Text

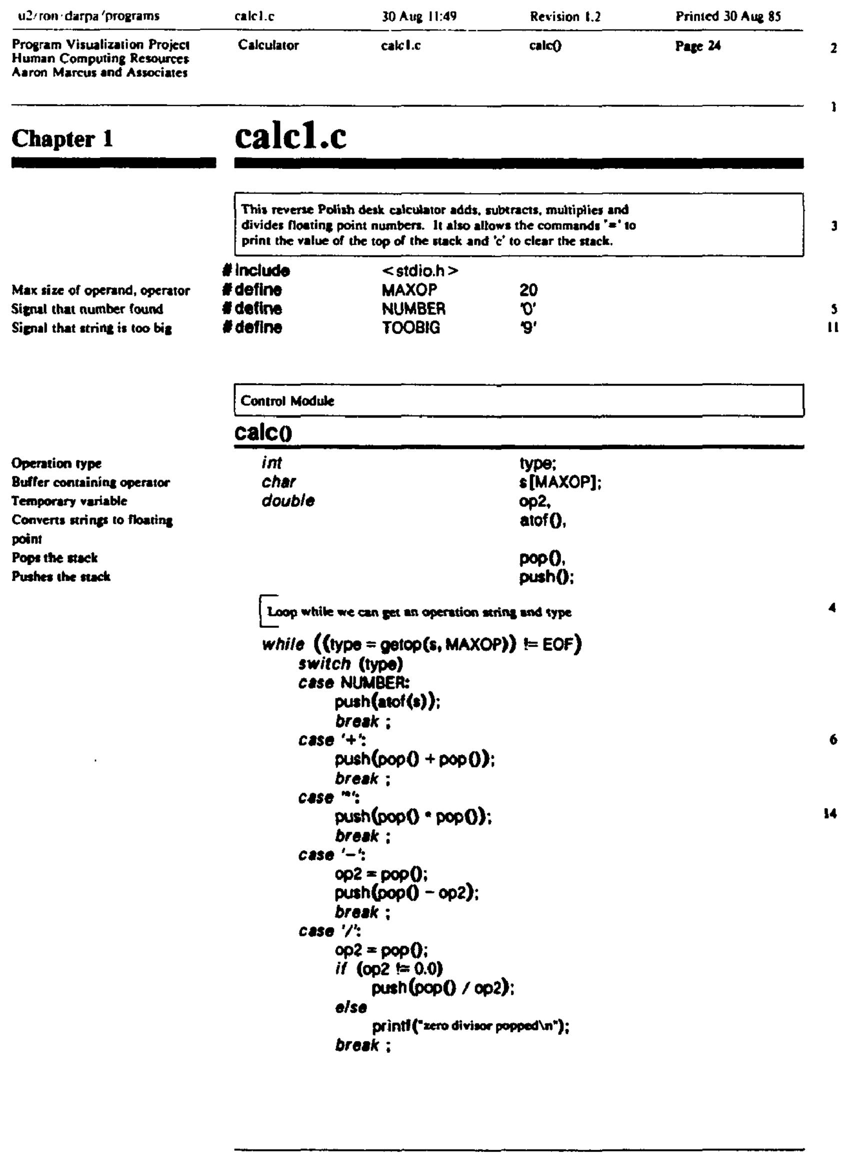
We illustrate our design by systematically applying it to an example program. The example consists of a slightly updated version of a desk calculator program that appears in a standard book on C [Kernighan & Ritchie, 1978].

Figure 1 is the second page of the program as it is output using conventional program listing software on a typical dot matrix line printer. Even the lightness and fuzziness of the type reflects an unfortunate aspect of the way most program listings are produced today.

Figure 2 shows the first three pages of the same C program as output by the current version of the SEE processor driving a laser printer equipped with an appropriate set of fonts. The C program was not modified at all for input to SEE. The SEE output was massaged only in the introduction of some white space, since good white space introduction and good pagination are not yet done by SEE. Specific aspects of the program presentation are described below in numbered points (e.g., #1). Except where noted, the numbers refer to specific sections of Figure 2, and can be found in its right hand margin.

```
/*                    Stack Management Module                      */

#define MAXVAL 100        /* maximum depth of val stack */

int sp = 0;               /* stack pointer */
double val[MAXVAL];       /* value stack */

double push(f)            /* push f onto value stack */
double f;
{
        if (sp < MAXVAL)
                return (val[sp++] = f);
        else {
             printf("error: stack full\n");
             clear();
             return(0);
        }
}

double pop()              /* pop top value from stack */
{
        if (sp > 0)
                return(val[--sp]);
        else {
             printf("error: stack empty\n");
             clear();
             return(0);
        }
}

clear()                   /* clear stack */
{
        sp = 0;
}

/*                        Input Module                             */

getop(s, lim)             /* get next operator or operand */
char s[];                 /* operator buffer */
int lim;                  /* size of input buffer */
{
        int i, c;

        /* skip blanks, tabs and newlines */

        while ((c = getch()) == ' ' || c == '\t' || c == '\n')
                ;

        /* return if not a number */

        if (c != '.' && (c < '0' || c > '9'))
                return(c);
        s[0] = c;

        /* get rest of number */

        for (i = 1; (c = getchar()) >= '0' && c <= '9'; i++)
                if (i < lim)
                        s[i] = c;
```

Figure 1



u2/ron·darpa/programs calc1.c 30 Aug 11:49 Revision 1.2 Printed 30 Aug 85

Program Visualization Project
Human Computing Resources
Aaron Marcus and Associates

Calculator calc1.c calc() Page 24 2

1

Chapter 1 calc1.c

This reverse Polish desk calculator adds, subtracts, multiplies and divides floating point numbers. It also allows the commands '=' to print the value of the top of the stack and 'c' to clear the stack. 3

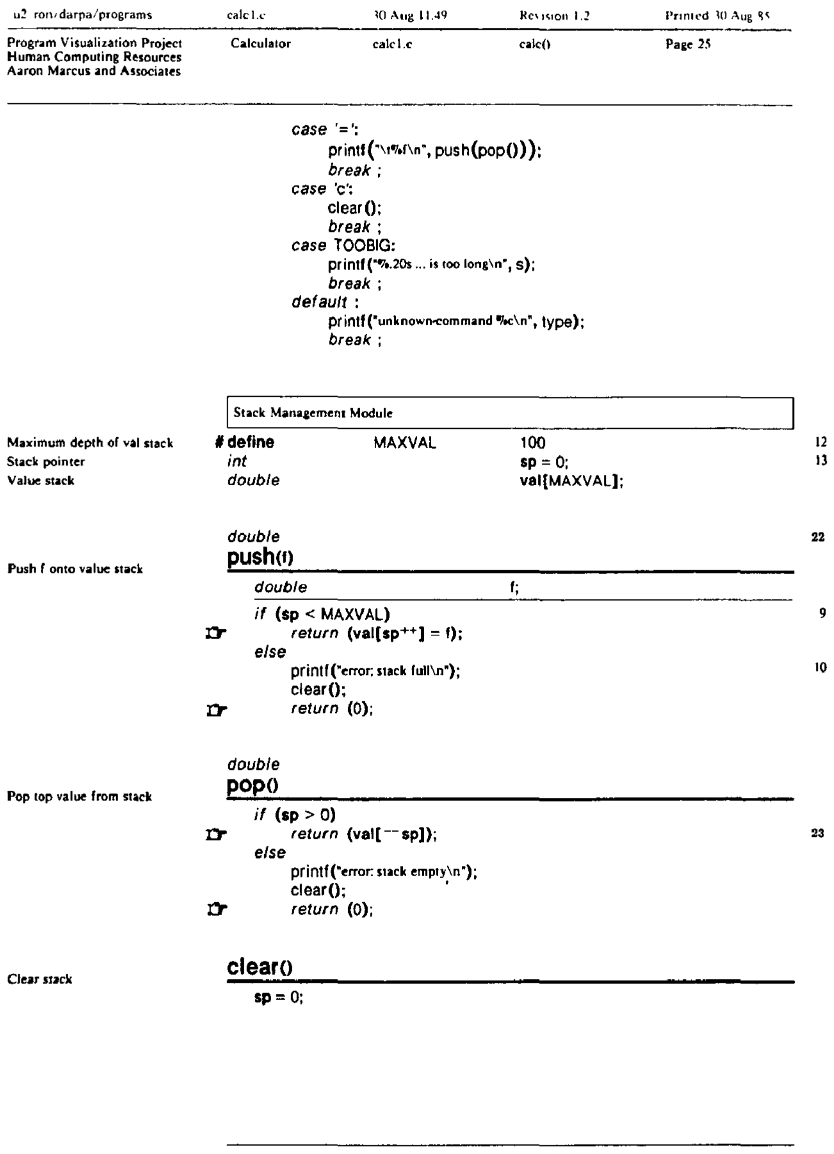
```
                                   #include   <stdio.h>
Max size of operand, operator      #define    MAXOP      20
Signal that number found           #define    NUMBER     '0'     5
Signal that string is too big      #define    TOOBIG     '9'     11
```

Control Module

calc()

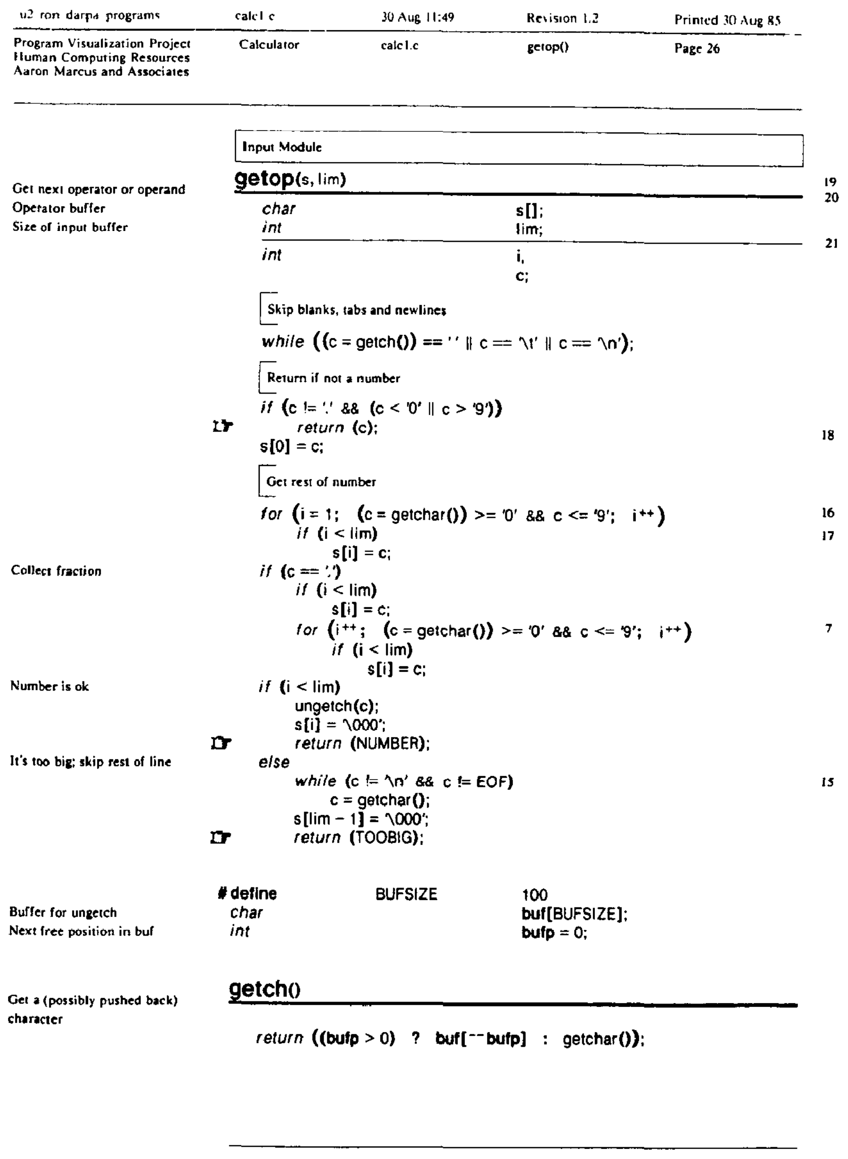
```
Operation type                       int        type;
Buffer containing operator           char       s[MAXOP];
Temporary variable                   double     op2,
Converts strings to floating                    atof(),
point
Pops the stack                                  pop(),
Pushes the stack                                push();

   Loop while we can get an operation string and type                  4
   while ((type = getop(s, MAXOP)) != EOF)
       switch (type)
       case NUMBER:
           push(atof(s));
           break ;
       case '+':                                                        6
           push(pop() + pop());
           break ;
       case '*':
           push(pop() * pop());                                         14
           break ;
       case '-':
           op2 = pop();
           push(pop() - op2);
           break ;
       case '/':
           op2 = pop();
           if (op2 != 0.0)
               push(pop() / op2);
           else
               printf("zero divisor popped\n");
           break ;
```

Figure 2a



```
u2 ron/darpa/programs         calc1.c      30 Aug 11.49   Revision 1.2   Printed 30 Aug 85
Program Visualization Project Calculator   calc1.c        calc()         Page 25
Human Computing Resources
Aaron Marcus and Associates

                               case '=':
                                    printf("\t%f\n", push(pop()));
                                    break ;
                               case 'c':
                                    clear();
                                    break ;
                               case TOOBIG:
                                    printf("%.20s ... is too long\n", s);
                                    break ;
                               default :
                                    printf("unknown-command %c\n", type);
                                    break ;

                             Stack Management Module
Maximum depth of val stack   #define      MAXVAL        100                   12
Stack pointer                int                        sp = 0;               13
Value stack                  double                     val[MAXVAL];

                             double                                           22
Push f onto value stack      push(f)
                               double                 f;
                               if (sp < MAXVAL)                               9
                           ☞        return (val[sp++] = f);
                               else
                                    printf("error: stack full\n");            10
                                    clear();
                           ☞        return (0);

                             double
Pop top value from stack     pop()
                               if (sp > 0)
                           ☞        return (val[--sp]);                       23
                               else
                                    printf("error: stack empty\n");
                                    clear();
                           ☞        return (0);

Clear stack                  clear()
                               sp = 0;
```

Figure 2b



```
u2 ron darpa programs         calc1 c      30 Aug 11:49   Revision 1.2   Printed 30 Aug 85
Program Visualization Project Calculator   calc1.c        getop()        Page 26
Human Computing Resources
Aaron Marcus and Associates

                              Input Module
Get next operator or operand  getop(s, lim)                                          19
Operator buffer                 char                    s[];                         20
Size of input buffer            int                     lim;
                                int                     i,                           21
                                                        c;

                                Skip blanks, tabs and newlines
                                while ((c = getch()) == ' ' || c == '\t' || c == '\n');

                                Return if not a number
                                if (c != '.' && (c < '0' || c > '9'))
                          ☞          return (c);                                     18
                                s[0] = c;

                                Get rest of number
                                for (i = 1;  (c = getchar()) >= '0' && c <= '9';  i++)   16
                                     if (i < lim)                                    17
                                          s[i] = c;
Collect fraction                if (c == '.')
                                     if (i < lim)
                                          s[i] = c;
                                     for (i++;  (c = getchar()) >= '0' && c <= '9';  i++)  7
                                          if (i < lim)
                                               s[i] = c;
Number is ok                    if (i < lim)
                                     ungetch(c);
                                     s[i] = '\000';
                          ☞          return (NUMBER);
It's too big; skip rest of line else
                                     while (c != '\n' && c != EOF)                   15
                                          c = getchar();
                                     s[lim - 1] = '\000';
                          ☞          return (TOOBIG);

                              #define          BUFSIZE          100
Buffer for ungetch              char                            buf[BUFSIZE];
Next free position in buf       int                             bufp = 0;

Get a (possibly pushed back)  getch()
character
                                return ((bufp > 0)  ?  buf[--bufp]  :  getchar());
```

Figure 2c

(#1) (Fig. 2a) The program is presented on a standard 8½x11 inches page that is separated into four regions, a header, a footnote area, a code column, and a marginalia comment column.

The Presentation of Program Environment

A full understanding of a program cannot come from reading only the code, but requires a knowledge of the context in which the program was created and is used.

(#2) (Fig. 2a) The header of each program page therefore contains important metadata about the program environment, namely the title, the file being listed, the location of the file within the directory system of the computer, the function which is being defined at the top of the page, the date and time the file was last changed, an optional revision number, the date that the listing was made, and the page number within the listing.

The Spatial Composition of Comments

The process of creating comments and integrating them effectively with code is not facilitated by the programming methodologies, text editors, and development environments commonly available. We have therefore focused on methods for presenting comments for maximum effect, both in isolation and in relationship to code.

(#3) (Fig. 2a) Comments that are *external* to function definitions are displayed in a small-sized serif font inside an outline box. There is ample margin allowance around the text to ensure optimum legibility and readability.

(#4) (Fig. 2a) Comments *internal* to function definitions are displayed in a small-sized serif font appropriately indented and marked by a left vertical bracket.

(#5) (Fig. 2a) Comments located on the same lines as source code, which we call *marginalia* comments, are displayed in a small-sized serif font in the marginalia column. These items should be short single line phrases.

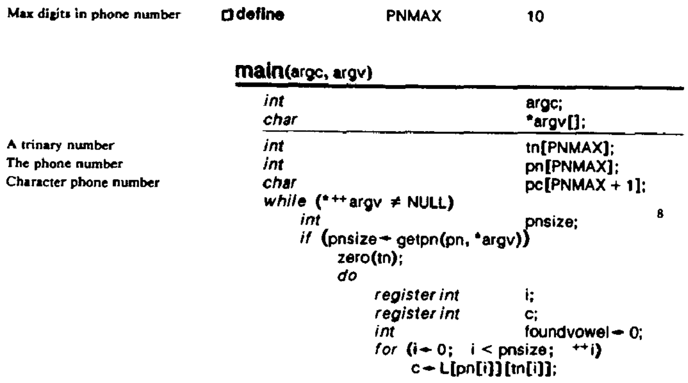
The Typography of Program Punctuation

Computer program punctuation marks consist of separators such as ";" and ".", containment symbols such as "(" and "}", and operators such as ".", "!", and "!=". Their legibility is a critical component affecting the comprehensibility of a program, much more so than the legibility of English language punctuation affects the comprehensibility of a passage in English.

(#6) (Fig. 2a) In this example the ";" appears in 10 point regular Helvetica type, and thus uses the same typographic parameters as does much of the program code. The ":", on the other hand, has been set in bold type, and the "," has been enlarged to 14 point. (For the ",", look up approximately 5 lines to the *while* statement.) These distinctions highlight the difficulties in achieving legible punctuation with currently available typefaces. The bold is often slightly too heavy; the regular weight is sometimes degraded through photocopying. In addition, idiosyncratic size changes for particular characters in particular fonts are often desirable.

(#7) (Fig. 2c) Symbols such as the "++" and the "− −" have been kerned, that is, the letter spacing of individual characters has been overlapped to make them more readable. We have also adopted the perhaps controversial technique of displaying all unary operators as superscripts.

(#8) (Fig. 3) In C, there are many symbols that clearly could benefit from improved appearance, e.g., "=", ">=", "==", and "!=". We could define an improved symbol set that would be substituted for the conventional characters. Figure 3, for example, illustrates the use of a small number of symbol substitutions, namely, an open square in place of the "#" character, a left-pointing arrow in place of the "=", the assignment operator, and a mathematical "not equals" sign in place of "!=". Whether or not such substitutions are to be invoked should be determined by a flag under control of the user. Legibility criteria would suggest innovation; however, reader familiarity and direct semantic reference to two input keyboard strokes would suggest the conventional alternative.



```
Max digits in phone number   □define        PNMAX           10

                             main(argc, argv)
                                int                         argc;
                                char                        *argv[];
A trinary number                int                         tn[PNMAX];
The phone number                int                         pn[PNMAX];
Character phone number          char                        pc[PNMAX + 1];
                                while (*⁺⁺argv ≠ NULL)
                                   int                      pnsize;          8
                                   if (pnsize ← getpn(pn, *argv))
                                      zero(tn);
                                      do
                                         register int       i;
                                         register int       c;
                                         int                foundvowel ← 0;
                                         for (i ← 0;  i < pnsize;  ⁺⁺i)
                                            c ← L[pn[i]][tn[i]];
```

Figure 3

Typographic Encodings of Token Attributes

Current attempts at program visualization often employ crude mechanisms for distinguishing typographically one kind of token from another. Reserved words are often shown in bold face, a convention that began with Algol and has persisted despite its inappropriateness. For professional programmers, the reserved words do not need to be highlighted or strengthened! This attempt, however, typical of many prettyprinting programs, represents but one of the wealth of the purely typographic possibilities for enhancing the legibility and readability of programs. The optimum encoding is a complex synthesis of the reader's needs for clarity both when scanning the text quickly and when examining it slowly.

(#9) (Fig. 2b) Most tokens are shown in a regular sans-serif font; reserved words are shown in italic sans-serif type. Bold sans-serif is used to highlight global (*extern*) variables (see #23).

(#10) (Fig. 2b) String constants are shown appropriately de-emphasized in a small-sized serif font.

The Presentation of Preprocessor Commands

The lexical structure of C encodes all preprocessor commands with a prepended "#". In addition, a standard convention for C programming is the use of all capitalized letters to differentiate preprocessor identifiers (such as manifest constants) from all other tokens.

(#11) (Fig. 2a) The "#" signifying a preprocessor command is exdented to enhance its distinguishability from ordinary C source text.

(#12) (Fig. 2b) Macros (preprocessor identifiers) and their values are presented at two appropriate horizontal tab positions, an 8 pica tab stop and a 16 pica tab stop.

Even stronger methods may be used to separate visually the statements in the "preprocessor language" from those of C proper, for example, applying a gray tone to all the preprocessor statements.

The Presentation of Declarations

Much of a program's intractability often occurs in the declaration of variables as instances of particular data types and the initialization specifying values for certain variables.

(#13) (Fig. 2b) Identifiers being declared are aligned to a single implied vertical line located at an appropriate horizontal tab position, the 16 pica tab stop.

An additional factor that affects the readability of declarations with supporting comments is their tendency to cluster in long sequences. An alternative to the presentation method used in Figure 2 is shown in Figure 4.

```
Really a union!________________typedef struct marital {
                                   struct human {
Secs past millenium___________________int                    age;
M/f/n yes/no__________________________int                    sex;
Canadian dollars!_____________________int                    income;
Alive, dead, decaying_________________int                    health;
A list________________________________struct activity        *interests;
The main squeeze______________________struct human           *mate;
Your younger sibling__________________struct human           *sibling;
                                      }
                                                             husband;
                                   struct human              wife;
A queue____________________________struct human              *rug_rats;
                                   int                       status;
                                   }
                                                                      MARRIAGE;
In the beginning_______________struct human                           adam;
```

Figure 4

An even more difficult problem is that of enhancing the clarity of structure definitions. We suppress C's ineffective curly braces and substitute a much more compelling diagrammatic convention in Figure 5.

```
                         typedef struct marital
Really a union!          ┌
                           struct human
                           ┌
Secs past millenium           int                age;
M/f/n yes/no                  int                sex;
Canadian dollars!             int                income;
Alive, dead, decaying         int                health;
A list                        struct activity    *interests;
The main squeeze              struct human       *mate;
Your younger sibling          struct human       *sibling;
                           └
                                                 husband;
                           struct human          wife;
A queue                    struct human          *rug_rats;
                           int                   status;
                         └
                                                          MARRIAGE;
In the beginning         struct human                     adam;
```

Figure 5

The Visual Parsing of Expressions

One of the most difficult aspects of reading computer programs is parsing complex expressions. C has 46 different operators occurring at 16 levels of precedence, some associating left to right, others associating right to left. There is usually no help provided to the reader trying to decipher an expression other than where nesting and grouping is shown through the inclusion of parentheses. The resulting visual clutter and masking of what is essential is readily apparent in languages such as LISP.

(#14) (Fig. 2a) Parentheses and brackets are emboldened to call attention to grouped items. Nested parentheses are varied in size to aid the parsing of the expression.

(#15) (Fig. 2c) The word spacing between operators within an expression is varied to aid the visual parsing of the expression. Operands are positioned closer to operators of high precedence than to operators of low precedence.

The power of the combined effects of varying the word spacing, varying the parenthesis size, and kerning and raising the unary operators can be seen in the even more dramatic and controversial Figure 6.

```
m = e + k;
m = e++ + k;
m  =  e  +  k * k;
m  =  e  +  k * k  -  m * m;
m  =  e++  +  k * k  -  m * m;
m   =   e++  +  k * k   >>   2  -  m * m;
m = e++ >= 0;
m    =    e++ >= 0   &&   k * k  >>  2   !=  0;
m    =    e++ >= 0  &&  k * k >> 2 != 0   ||   m * m  >  0;
m = (e + k) * k;
m = (e + k) * (k - m) * m;
m = (e++ + k) * (k - m) * m;
m = (e++ + k) * (k >> (2 - m)) * m;
m = e++ >= ((0 && k * k) >> (2 != 0));
m    =    (e++ >= 0)  &&  k * (k >> 2)  !=  0   ||   (m * m) > 0;
for (a = 1 , e = 10;   a <= 10;   ++a, --e)
      a  +=  a * e  +  *p1;
**p[2] = sig(((*func(&a)) + ((a * b) + (b * c)) / 12)) + 2;
```

Figure 6

The Visual Parsing of Statements

Statements within a C program nest recursively. At any level, statements such as the *if*, *do...while*, and *switch* contain several component expressions or statements that must be parsed and understood in order that the statement as a whole may be understood. The resulting configuration of separate and nested statements presents a challenge to effective spatial structuring.

(#16) (Fig. 2c) Systematic indentation and placement of key words is employed as in conventional prettyprinting.

(#17) (Fig. 2c) Since curly braces are redundant with systematic indentation, they are removed in this example. Since this is controversial, whether it happens or not is controlled by a flag which may be set by the user.

(#18) (Fig. 2c) "Unusual" control flow is marked in the margin with pointing figures. A program visualizer should allow various definitions of "unusual", such as our current choice, which is any statement preceded by a label, any *goto* statement, any *continue* statement, any *break* statement not at the end of a *case*, any statement ending a *case* that is not a *break* statement, and any *return* statement that is not the final statement within a function definition.

The Presentation of Function Definitions

The presence of functions help determine for the reader the general sequence and rationale for the program's structure. Making these major "chunks" of the program immediately accessible can contribute significantly to the program's readability.

(#19) (Fig. 2c) The introductory text of a function definition, that is, the function name, is shown as a bold sans-serif headline.

(#20) (Fig. 2c) A heavy rule appears under the introductory text of a function definition.

(#21) (Fig. 2c) A light rule appears under the declaration of the formal parameters.

(#22) (Fig. 2b) The function type specifier, indicating the type of the value returned by the function, is placed on a line of its own above the function name. One reason this is done is to allow the program reader to find the names of all defined functions by scanning down the left margin of the code column and looking for the headlines.

The Presentation of Program Structure

A C program consists of one or more C source files. Each C source file contains some number of definitions of identifiers naming data items and functions. Source files also typically refer to a number of included external files which bind the external references to identifiers not defined within the current file. Identifiers which are defined in all functions of all files are known as global variables, and represent a major source of C programming errors.

(#23) (Fig. 2b) We therefore call attention to most uses of globals (but not manifest constants) by highlighting them in bold face.

We also can include as footnotes to the source text cross-references relating global variables appearing on that page to the location of their definitions or uses on other pages. (For an example of this, see pp. 65-67 of Volume 3 of [Baecker et. al. 1985].) The effect is that of a cross-reference index distributed in footnotes throughout the program.

Another aspect of program structure is its evolution over time. A program is not a static entity, but a series of approximations to an ultimate, perhaps never to be realized, state. We could use gray tone to highlight source code that has been changed since the "last version", or show with a strike-through line all text that has been removed, or use a carat to indicate the place from which text has been removed.

A Program Visualization Implementation

The SEE processor has been implemented as a modified version of PCC, the Portable C Compiler [Johnson, 1979].

Unlike Pass 1 of the original PCC, which sometimes directly generates assembly language, our first pass produces syntax trees for all C constructs. Comments are recognized and passed through instead of being stripped out. SEE's Pass 1 is also modified and supported by a special preprocessing pass in order to be able to handle C preprocessor statements.

The second pass has also been significantly modified in order to output instructions for the Device Independent TROFF document formatter [Kernighan, 1982] instead of assembler instructions. This output consists of C source code and comments interspersed with calls on TROFF macros that determine how code and comments are to be formatted. Most typographic variations are determined by table entries which allow SEE to be heavily parameter-driven. Thus it can produce a wide variety of program appearances under user control.

Summary, Conclusions, Suggestions for Future Research

We have presented in this paper ten fundamental areas of programming language functionality and several methods by which the presentation of the language constructs within each area can be enhanced. The methods are the result of the application of five basic graphic design principles of program visualization which are also presented in the paper:

- Principle I, Typographic Vocabulary, was applied in the typographic encodings of tokens, in the improvement of C punctuation marks, and in the kerning of multi-character operators.
- Examples of the application of Principle II, Typesetting Parameters, were the typesetting of file names and function names, the rules for the indentation within statements, and the typesetting of expressions, varying the word spacing around operators and the size of parentheses.
- Principle III. Page Composition, was used in the overall design of the program page, in the display of the three kinds of comments, and in the presentation of declarations and function definitions. The implicit rules defined by the left margin of the code column and the 8 pica and 16 pica tab stops are particularly important ways to guide the eye to the most important components within function definitions, declarations, and preprocessor macro definitions.
- Principle IV, Symbols and Diagrammatic Elements, was applied in the symbol substitutions and in the display of comments and of structures.
- Examples of Principle V, Metatext, are the header and the footer material, which assist the reader in navigating through his program and in maintaining and managing it within a computer system environment.

Enhanced program presentation is not without its difficulties. Listings are typically longer, although this problem can be solved with some loss of legibility by setting the type at a smaller point size. Visual compilers need to be highly parameterized, since optimum program appearance is a highly individualistic and contentious issue. Visual compilation is also expensive, although wasted programmer time is more expensive. Finally, we cannot yet prove with hard empirical evidence that these methods of display make programs more readable, comprehensible, and maintainable. We can only submit that the figures in this paper and the more comprehensive series which appears in [Baecker et. al., 1985] provide compelling visual evidence of the appropriateness and the efficacy of the proposed techniques.

There are a large number of remaining problems requiring future research:

• There are several areas fundamental to the enhanced presentation of program source text that we have not yet automated. These are the automatic introduction of white space, appropriate automatic line breaking, appropriate automatic page breaking, incorporation of programmer formatting intentions, display of pragmatics, display of additional diagrammatic representations [Martin & McClure, 1985], and comprehensive automatic warnings and annotations.

• Our work needs to extended to programming languages other than C. Although each new language will have its own peculiar problems, our methodology can clearly be applied across a wide family of languages.

• Even more interesting is the extension of the work to interactive program visualization, which offers a host of new opportunities to incorporate dynamics, animation, color, and sound. We are no longer faced with the difficult problem of establishing "the best" mapping between token types and typographic styles, for the program can be easily re-displayed with different settings. Even more significantly, we can depict through image dynamics and through animation features of the program *in execution* [Baecker, 1975; Baecker and Sherman, 1981; Brown and Sedgewick, 1984].

• There are a great many problems, beyond the scope of this paper, remaining to be solved before a system such as SEE can be implemented with ease and with elegance.

• There also remains the need to substantiate experimentally that these proposed methods of presentation are effective. We plan a series of experiments to measure if our display conventions make programs more readable, understandable, appealing, memorable, and maintainable, and, if so, by how much. Ultimately, there must be developed an information processing model of program reading which would serve as a scientific basis for answering such questions and for facilitating even further improvements in the art of computer program presentation.

Acknowledgements

This work was sponsored by the Defense Advanced Research Projects Agency under ARPA Order 4469 and by Human Computing Resources Corporation, Toronto. We are indebted for valued research assistance to Michael Arent, Paul Breslin, Bruce Browne, John Longarini, Allen McIntosh, David Slocombe, Chris Sturgess, and Tracy Tims, and to Brad Myers for some insightful comments on an early draft of this paper.

References and Bibliography

Baecker, R. (1975). Two systems which produce animated representations of the execution of computer programs. *SIGCSE Bulletin, 7* (1), February 1975, 158-167.

Baecker, R. & Sherman, D. (1981). *Sorting Out Sorting,* 30 minute colour sound film, Dynamic Graphics Project, Computer Systems Research Institute, University of Toronto, 1981. (Excerpted in *SIGGRAPH Video Review 7,* 1983.)

Baecker, R. & Marcus, A. (1983). On enhancing the interface to the source code of computer programs. *Proc. Human Factors in Computing Systems* (SIGCHI '83), Boston, December 1983, 251-255.

Baecker, R., Marcus, A., Arent, M., Longarini, J., Macintosh, A., and Tims, T. (1985) *Enhancing the Presentation of Computer Program Source Text,* Final Report of the Program Visualization Project to the Defense Advanced Research Projects Agency, submitted by Human Computing Resources Corporation and Aaron Marcus and Associates, 6 volumes, 1985.

Brown, M.H. & Sedgewick, R. (1984). A system for algorithm animation. *Computer Graphics, 18* (3), 1984, 177-186.

Grogono, P. (1979). On layout, identifiers and semicolons in pascal programs. *SIGPLAN Notices, 14* (4), 35-40.

Gustafson, G.G. (1979). Some practical experiences formatting pascal programs. *SIGPLAN Notices, 14* (9), 42-49.

Harbison, S.P. & Steele, Jr., G.L. (1984). *C: A reference manual.* Prentice-Hall, Inc.

Hueras, J. & Ledgard, H. (1977). An automatic formatting program for pascal. *SIGPLAN Notices, 12* (7), 82-84.

Johnson, S.C. (1979). A tour through the Portable C Compiler. *UNIX Programmer's Manual Volume 2.*

Kernighan, B. & Ritchie, D. (1978). *The C programming language.* Prentice-Hall, Inc.

Kernighan, B. (1982). A typesetter-independent TROFF. *Bell Laboratories Computing Science Series Technical Report No. 97,* March 1982.

Leinbaugh, D. (1980). Indenting for the compiler. *SIGPLAN Notices, 15* (5), 41-48.

Marcus, A. & Baecker, R. (1982). On the graphic design of program text. *Proceedings of Graphics Interface 82,* 302-311.

Martin, J. & McClure, C. (1985). *Diagramming techniques for analysts and programmers.* Englewood Cliffs, NJ: Prentice-Hall, Inc.

Miara, R.J., Musselman, J.A., Navarro, J.A. & Schneiderman, B. (1983). Program indentation and comprehensibility. *Comm. of the ACM, 26* (11), 861-867.

Oppen, D.D. (1980). Prettyprinting. *ACM Transactions on Programming Languages and Systems, 2* (4), 465-483.

Rose, G.A. & Welsh, J. (1981). Formatted programming languages. *Software -- Practice and Experience, 11,* 651-669.

Rubin, L. (1983). Syntax-directed pretty printing -- A first step towards a syntax-directed editor. *IEEE Transactions on Software Engineering, 9* (2), 119-127.