MH

1543

Bell Laboratories Cover Sheet for Technical Memorandum

The information contained herein is for the use of employees of Bell Laboratories and is not for publication. (See GEI 13.9-3)

Title- **Beautifying C Code**

Date- **April 15, 1980**

TM- **80-1271-4**

Other Keywords- **Programming style**

Author	Location	Extension	Charging Case- **39199**
L. Cherry	**2c516**	**6067**	Filing Case- **39199-11**

ABSTRACT

For a program to be easy to understand, it must be readable. The best method of enhancing program readability in free format languages like C is to use indentation to reflect the nesting level of the code. The program **cb** automatically adds such indentation. **cb** has two levels of operation. At the first level it enforces a minimum of rules, allowing the user to maintain a great deal of individuality in code layout. At the second level, it enforces many layout rules, putting the code in a canonical form. This strict level of operation makes **cb** useful in large projects for enforcing standards. In this paper I will discuss common layout conventions for C code, design criteria for the beautifier, and how it works.

Pages Text 10	Other 1	Total 11
No. Figures 0	No. Tables 0	No. Refs. 3

DISTRIBUTION
(REFER GEI 13.9-3)

COMPLETE MEMORANDUM TO

CORRESPONDENCE FILES

OFFICIAL FILE COPY
PLUS ONE COPY FOR
EACH ADDITIONAL FILING
CASE REFERENCED

DATE FILE COPY
(FORM E-1328)

10 REFERENCE COPIES

<ANO,ALFRED V
<BALENSON,CHRISTINE M
<BARON,ROBERT V
<BECKER,RICHARD A
BROWN,W STANLEY
<CHEN,STEPHEN
<CHERRY,LORINDA L
<FELDMAN,STUART I
<FRASER,A G
<GOLDSTEIN,A JAY
<GRAHAM,R L
+HANNAY,N B
<JOHNSON,STEPHEN C
<KEESE,W M
<KERNIGHAN,BRIAN W
<LUDERER,GOTTFRIED W R
<MARANZANO,J F
<MARKY,GERALDINE A
+MC DONALD,H S
<MC GILL,R
MCILROY,M DOUGLAS
<MENNINGER,R E
<MORGAN,SAMUEL P
+PEIM,R C
<RALEIGH,T M
<RIDDLE,GUY G
<SCHLEGEL,C T
<STORER,JAMES A
+STREETER,LYNN A
<SZYMANSKI,THOMAS G
TERRY,MILTON E
<WEINBERGER,PETER J
+WONSIEWICZ,S C
<YANNAKAKIS,MIHALIS
34 NAMES

COVER SHEET ONLY TO

CORRESPONDENCE FILES

4 COPIES PLUS ONE
COPY FOR EACH FILING
CASE

AAGESEN,JOHN
ACKERMAN,J T
ALBERS,RAINER B
<AHUJA,SUDHIR R
ALDERALLA,RICHARD J
ALCALAY,D
ALKONS,FREDERICK
ALLEN,R C
AMITAY,N
ANDERSON,KATHRYN J
ANDERSON,MILTON M
ARNDT,DENNIS L
ARNOLD,GEORGE W
ARNOLD,JAMES C
ARNOLD,PHYLLIS A
ARNOLD,THOMAS F
ARZBERGER,C R
ASELTINE,EDWARD G
AULL,DENIS W
BAGGA,YUDHVEER S
BAKER,BRENDA S
BALLANCE,ROBERT A
BALLARD,E D,JR
BARBATO,ROBERT E
BARONSKY,ALLEN
BAUER,BARBARA T
BAUER,HELEN A
<BAUGH,C R
BEACH,MILTON
BEAUMONT,LELAND R
BEBLO,WILLIAM
BEDNAR,JOSEPH A,JR
BENCO,DAVID S
BENISCH,JEAN
BENNETT,RAYMOND W
BENNETT,WILLIAM C
BERGLAND,G D
BERNHARDT,RICHARD C
BERNSTEIN,L
BERZINS,ALEXANDER H
BEYLER,ERIC
BICKFORD,NEIL B
BILONOS,R M
<BIREN,IRMA B
BISHOP,J DANIEL
BISHOP,THOMAS P
BITTNER,B B
BLAKE,GARY D
BLAZIER,S D
BLECHMAN,RONALD I
BLEIER,JOSEF
BLINN,J C
BLOSSER,PATRICK A
BLUM,MARION
BODEN,F J
BOEHM,KIM R
BOGART,THOMAS G
BOIVIE,RICHARD H
BOLSKY,MORRIS I
BORISOV,ELLEN A
BOURNE,STEPHEN R
BOWYER,L RAY
<BOYCE,W M
BOYER,PHYLLIS J
BOYLE,GERALD C
BRADLEY,M HELEN
<BRANDT,RICHARD B
BRIGGS,GLORIA A
BROAD,MARINA M
BROOKS,CATHERINE ANN
BROSS,JEFFREY D
BROWN,ELLINGTON L
BROWN,W B
BRUECKNER,DOUGLAS E
BUIST,L W
BURG,F M
<BURNETT,DAVID S
BURCFF,STEVEN J
BURROWS,THOMAS A
BUTLETT,DARRELL L
BYERLEE,R W
BYRNE,EDWARD R
CABLE,GORDON G,JR
CALESSO,GIULIO L
CAMPBELL,JERRY H
<CANADAY,RUDD H
CARLSON,HELEN V
CARTER,DONALD H
CASPERS,BARBARA E
CASTELLANO,MARY ANN
CAVINESS,JOHN D
CERMAK,I A
CHAI,D T
CHAMBERS,B C
<CHAMBERS,J A
CHANEY,F W
CHANG,JEAN L
CHANG,JO-MEI
CHAPPELL,S G
CHENG,Y
CHEN,PING C
CHESSON,GREGORY L
CHILDS,CAROLYN
CHI,M C
CHODROW,M M
CHONG,PHEE
CHRIST,C W,JR
CIEMINSKI,DEBRA F
CLARK,DAVID L
CLARY,JOHN B
CLAYTON,D P
CLINE,LAUREL M I
COHEN,HARVEY
COHEN,RICHARD L
<COLE,LOUIS A
COLLICOTT,E B
CONDON,J H
CONKLIN,DANIEL L
CONNERS,RONALD E
COOK,JOEL M
COOK,T J
COOPER,ARTHUR E
COSTELLO,J W
COTTRELL,JENNIE I
COVINGTON,RALPH L
CRAGUN,JOAN
CRAIG,JOHN B
CRISTOFOR,EUGENE
<CROME,L L
CSURI,JOSEPH A
CSENCSITS,BRENDA M
DAGNALL,C H,JR
DAVEY,DOUGLAS A
DAVIS,B DREW
DAWSON,JOHN E
DAY,P W
DE PAZIO,M J
DE GRAAF,D A
DE TREVILLE,JOHN D
DEAN,JEFFREY S
DELLNER,R J
DENKMANN,W JOHN
DENNY,MICHAEL S
DENSMORE,SUSAN
DESMOND,JOHN PATRICK
DEVLIN,SUSAN J
DI PIETRO,R S
DIB,GILBERT
DIMARCO,ROBERT T
<DIMMICK,JAMES O
DINEEN,THOMAS J
DITZEL,DAVID R
DODLINZ,BARBARA ANN
DOLATOWSKI,VIRGINIA M
DOLOTTA,T A
DOMANGUE,JAMES C,JR
DOWDEN,DOUGLAS C
DRAKE,LILLIAN
DREIZLER,H K
D'ANDREA,LOUISE A
DUDICK,ANTHONY
DUGGER,DONALD D
DUMAIS,VALERIE
<DUNCANSON,ROBERT L
DUNKIN,PATRICIA
DUYER,T J
DYER,MARY E
EDMUNDS,T W
EICHORN,KURT H
<EITELBACH,DAVID L
EKSTROM,SUSAN
ELDRIDGE,JOHN
ELLIOTT,RUBY J
ELLIS,DAVID J
ELY,T C
EPLEY,ROBERT V
EPSTEIN,ROBERT H
ESSERMAN,ALAN R
EVANS,MELVIN J
FIERMAN,THOMAS L,JR
FABISCH,M P
FABRICIUS,WAYNE N
FAIRCHILD,DAVID L
FAULKNER,ROGER A
FELDER,J
FERREK,NANCY L
FLOSTER,I REED
<FICHTNER,WOLFGANG
FILDES,NEAL R
FISCHER,HERBERT B
FISHMAN,DANIEL A
FLANDRENA,A
FLEISCHER,A I
FLEMING,JAMES R
FLOWER,ELLEN F
FONG,K T
FORTNEY,V J
FOUGHT,B T
FOUNTOUKIDIS,A
FOWLER,BRUCE R
FOWLER,GLENN D
FOX,PHYLLIS A
FOY,J C
FRANKLIN,JAMES F
FRANK,AMALIE J
FREEMAN,K G
FREEMAN,MARTIN
FRENCH,A F,JR
FREY,R C
FRIED,LAWRENCE A
FROST,H BONNELL
FRUCHTMAN,BARRY
FULLERTON,L WAYNE
FU,C
GABBE,JOHN D
GALGOWSKI,ANN L
GARMISE,ROBERT
GARRISON,GARY A
<GATES,G F
GEARY,M J
GEEES,T J,JR
GEEE,E W
GEORGEN,MICHAEL R
GEPNER,JAMES R
GERSHMAN,ANATOLI V
GETLING,P T
GIBB,KENNETH R
GIBSON,J C
GIESKE,WILLIAM E
GILKEY,THOMAS J
<GILLETTE,DEAN
GIMPEL,J F
<GITHENS,J A
GITHENS,JAY L
<GLASSER,ALAN L
GLUCK,F G
<GLUSHKO,ROBERT J
<GNANADESIKAN,R
GOFF,CAROLYN E
GOLABEK,RUTH T
GOLDENBERG,H R
<GORMAN,J E
GORTON,D R
GOTTDENKER,R
GREENLAW,R L
GROSS,ALAN M
GROSS,ARTHUR G
GROSS,T M
GRUENWALD,JOHN
GRZELAKOWSKI,MARPEEN E
GUBITOSI,LOUIS E
...
695 TOTAL

+ NAMED BY AUTHOR > CITED AS REFERENCE < REQUESTED BY READER (NAMES WITHOUT PREFIX WERE SELECTED USING THE AUTHOR'S SUBJECT OR ORGANIZATIONAL SPECIFICATION AS GIVEN BELOW)

MERCURY SPECIFICATION.....

COMPLETE MEMO TO:
127-SUP

COVER SHEET TO:
12-DIR 13-DIR 127

COPRDS = COMPUTER PROGRAM DOCUMENTATION AND STANDARDS

TO GET A COMPLETE COPY:

1. BE SURE YOUR CORRECT ADDRESS IS GIVEN ON THE OTHER SIDE.
2. FOLD THIS SHEET IN HALF WITH THIS SIDE OUT AND STAPLE.
3. CIRCLE THE ADDRESS AT RIGHT. USE NO ENVELOPE.
4. INDICATE WHETHER MICROFICHE OR PAPER IS DESIRED.

NO CORRESPONDENCE FILES
NO TA127

TM-80-1271-4
TOTAL PAGES 11

PLEASE SEND A COMPLETE

( ) MICROFICHE COPY ( ) PAPER COPY

TO THE ADDRESS SHOWN ON THE OTHER SIDE.

Bell Laboratories

Subject: **Beautifying C Code**
Case- 39199 -- File- 39199-11

date: **April 15, 1980**

from: **L. Cherry**

TM: **80-1271-4**

MEMORANDUM FOR FILE

1. Introduction

The most important attribute a computer program can have, other than being bug-free, is to be easy to read and understand, and, therefore, easy to maintain and debug. Free format languages like C and Pascal allow the user great flexibility in program layout to enhance readability. The compilers have no built-in restrictions on code layout. Conventions have developed to display the structure of the code through indentation. Each level of nesting is displayed visually by the level of indentation. Unfortunately, the same flexibility that allows layout to enhance readability, also allows the user to totally obscure the structure of the code with its layout. Although the structure of poorly formated code may be obvious to the author, it is usually less than obvious to her/his colleague, who later must maintain the program. Even if the code is formated to show its structure, it may follow a different convention than the one to which the maintainer is accustomed. Although one solution to the problem of different conventions is for each programmer to have her/his own beautifier, another, more universal, solution is to have one program that puts the code in some agreed upon form. The program **cb** may be used to put a program in a standard form, and should help programmers who inherit programs from others.

cb was originally written with the philosophy that it should improve program readability by adding proper indentation, while allowing the code to retain all other features of individual layout style. This flexibility allowed users to add as much (or as little) other white space to the code as they wished to enhance readability. It also allowed several different conventions for placement of curly brackets. This philosophy is fine for individual users maintaining their own code but is too flexible for large projects in which programs are passed among many programmers. Such projects usually develop strict standards for code layout that all programmers are required to follow. However, defining standards and enforcing them are two very different problems. To accommodate the large software projects using C, **cb** has a second level of beautification, I'll call *strict*. In strict mode **cb** puts the code in a canonical form that conforms to the layout style used by Kernighan and Ritchie in "The C Program Language" [1].

In this paper I will discuss the design criteria for the beautifier, the different layout conventions used for C code, the algorithms used by **cb** and the problems these algorithms occasionally have.

2. Background and Design Criteria

The first C beautifier [2], which was never really used, was written with the philosophy that a beautifier's job is to produce a beautiful *listing* of a program. In addition to adding indentation, it capitalized keywords, added line numbers, and pointed out null statements. Other programs with this philosophy add numbers on the line to indicate nesting depth [3]. The philosophy of a program lister, the term "prettyprinter" has been coined, creates several problems for the user. First, the user cannot use the lister to put the actual code in a canonical form because the lister adds

characters to the code that are not part of the syntax of the language. Second, when debugging a program, the user is forced either to read two, possibly very dissimilar, versions of the code or to edit the real code to look like the beautified version. Third, programmers now debug and maintain code in a time sharing environment with context editors where the concept of line numbers is rarely important or used.

The C beautifier, the **cb** command on UNIX† , was written with the philosophy that the only job of a beautifier *is* to produce syntactically correct source code that adheres to a minimal number of layout rules that improve code readability. At the lowest level of beautification **cb** enforces two rules: (1) only one statement is allowed per line, and (2) each statement has a standard amount of indentation reflecting its nesting depth. Other than leading white space, all user layout style is preserved. This philosophy allows the user to write in an individual style but enforces "standard" indentation. In strict mode **cb** enforces many additional rules on white space and layout but only adds blanks, tabs and newlines. The code remains syntactically correct C.

After basic philosophy about the job of a beautifier, the next design issue is how to do that job. **cb** uses a heuristic approach to beautifying C code, getting clues to what it should do from the punctuation in the syntax of C. The most important punctuation characters are "{", "}", "(", ")", and ";". By using heuristics and adopting a press-on-regardless attitude, **cb** also becomes useful in the development stage of a program. The user can run the beautifier on a program that is not syntactically correct C and see where the indentation goes awry. Errors like missing closing brackets are often hard to find and are discovered by the compiler some distance from where the user intended the bracket. The visual blocking of the code done by **cb** frequently makes the missing bracket very apparent. One problem with using punctuation to beautify a program is that punctuation hidden from **cb** in preprocessor statements will cause errors in indentation. Although it would be nice to handle this problem, since **cb** only adds white space in non-damaging places, the consequences of being wrong are simply that the indentation may be wrong. If the input to **cb** would compile, so will the output.

Before presenting the heuristics **cb** uses in detail, a brief discussion of variability in C code layout might be useful. The following questions have interesting and surprising answers:

What is "standard" indentation?
It depends on whom you ask.

How many different styles of C layout are there?
Not quite as many as there are C programmers!

The next two sections will discuss some common conventions in indentation and layout.

3. Indentation Conventions

There are three different conventions for general indentation. The first, used by **cb** , follows that of Kernighan and Ritchie. With a few exceptions, each level of curly bracket causes one more level of indentation. Also, simple consequents of control statements, if they appear on a separate line, are indented one level more than the control statement. The following example from [1] illustrates this style:

```
main()
{
        int len;

        max = 0;
        while ((len = getline()) > 0)
                if (len > max) {
```

† UNIX is a Trademark of Bell Laboratories.

```
                        max = len;
                        copy();
                }
        if (max > 0)
                printf("%s",save);
}
```

Although this indentation style is very common, it is by no means universal among C programmers. Examination of the source code for UNIX commands shows two other indentation conventions.

The second style indents everything one level less than the first. In this style all code in a function at the first level begins on the left margin and all further nesting is reflected in the indentation from there. Programmers who write in this style claim it is easy to read and keeps the code from migrating too far to the right. Programmers who do not use this style complain of having difficulty finding global variable definitions, function definitions, and labels. The previous example in this style is:

```
main()
{

int len;
max = 0;
while ((len = getline()) > 0)
        if (len > max) {
                max = len;
                copy();
        }
if (max > 0)
        printf("%s",save);
}
```

The third style indents consequents of control statements one level more than the first. This style is usually accompanied by different treatment of curly brackets. Although this style is as easy to read as the first, the code does migrate quickly to the right, sometimes creating a line length problem. Our example in this style is:

```
main()
{
        int len;
        max = 0;
        while ((len = getline()) > 0)
                        if (len > max)
                                {
                                        max = len;
                                        copy();
                                }
        if (max > 0)
                        printf("%s",save);
}
```

Other conventions in indentation involve particular types of statements, rather than general program layout. One such statement is the `switch` statement, which has the three layout styles illustrated below.

```
switch (c) {
case 'a':
        i++;
```

```
        ...

switch (c) {
        case 'a':
                i++;
                ...

switch (c) {
    case 'a':
        i++;
        ...
```

In the last example the `case` is indented half a tab (4 spaces) from the `switch` and the `i++` is indented one full tab. Also connected with the `switch` is what I'll call the hidden `while` or `for` statement. This convention seems to be used when a major block of code is controlled by a `for` or `while` statement whose entire consequent is a giant `switch` statement. Here many programmers omit one level of indentation and write the `for` and `switch` on the same level as:

```
while ((type = getop(s)) != EOF)
switch (type) {
case '+':
        ...
```

This construction is sometimes written:

```
while ((type = getop(s)) != EOF) switch (type) {
case '+':
        ...
```

Another control statement that is sometimes hidden is the `else` in constructions like:

```
if (a)
        b; else
        c;
```

Here, if `b` is a complicated expression, it is easy to miss the conditional execution of `c`.

There are two styles of indentation for argument definitions in function definition.

```
func(arg1,arg2)
int arg1, arg2;
{
        ...
```

and

```
func(arg1,arg2)
        int arg1,arg2;
{
```

Finally, there are three different conventions for continuing statements across lines. First, the continued code is indented half an indent more than the current indent level; second, the continued code is indented one level more than the current level; last, the continued code is indented to the current level. cb uses the first convention because it seems to point out that the statement is continued, while not obscuring either the control statements themselves or their consequents.

4. Other Layout Conventions

After indentation the next area of differences in program layout in C is the location of curly brackets. The following conventions are used:

```
if (a) {
        b;

if (a)  {
        b;

if (a)
{
        b;

if (a)
        {
                b;
```

In the first three styles the matching closing bracket is usually lined up with the beginning of the `if`. In the last style the closing bracket is usually lined up with the opening bracket. In all styles the closing bracket may be on the line with the last statement in the block as in:

```
if (a){
        b;
        c; }
```

Finally, there are three conventions for placement of the body of control statements.

```
if(a){

if (a) {

if      (a) {
```

In the last style the open bracket is usually on the next line directly below the "(". The code inside the block is indented to the same level as the bracket. This style is hard to read because of the similarities between the characters "(" and "{" and the separation of the keywords from the statement body.

Some programmers use a combination of the above styles, using one for `for`, `while` and `switch` statements and another for `if-else` statements. The combination style is the hardest code to read because the rules for where to look for brackets depends on the type of statement in control of the block.

5. Methodology

The first approach to writing a C beautifier might be to use the first pass of the C compiler to parse the code. Although this approach has the advantage of guaranteeing that the beautifier would track the syntax of the language, it will not solve the problem. The code to be beautified is not C at all. The input to the beautifier is a combination of C and C preprocessor statements. It may not even be syntactically correct C. For example,

```
#define ever ;;

for (ever) {
```

is correct C only after the preprocessor makes the substitution for "ever". In the following more extreme example, the coding language can be defined by the preprocessor so that the code no longer even resembles C.

```
#define IF      if(
#define THEN    ){
#define ELSE    } else {
#define ELIF    } else if (
#define FI      ;}

IF a
THEN b;
ELIF c
THEN d;
ELSE e
FI
```

The use of preprocessor conditionals in the next example illustrates another problem

```
        func(a1, a2,
#ifdef A
        a3);
#else
        a4);
#endif
```

Of course a cleaner way to write this code would be

```
#ifdef A
        func(a1,a2,a3);
#else
        func(a1,a2,a4);
#endif
```

But cleaner or not the beautifier must be prepared to handle or at least recover from the code in either form.

Short of keeping all the punctuation from the define statements, which may be in `include` files, it is not possible to beautify code that redefines the language. Even if **cb** went to the trouble of resolving the `define` statements, the problem of what to do with preprocessor conditional code is even harder to resolve.

cb is designed to work almost entirely on C punctuation to beautify the code. Although C syntax is rich with *punctuation*, not *all* punctuation is used consistently in C syntax or in standard C layout. The main inconsistencies are:

1. ; ends a statement except inside a `for` statement.
2. { increases the indentation by one except inside `structure` initializations like:

```
struct a key = {
        { "if", 1 },
        { "while", 2 },
        ...
};
```

Here the outside curly brackets increase and decrease the indentation level, but the inside

curly brackets do not.

3. } decreases the indentation level by one and appears alone on a line except: 1) in `structure` initializations as described above, 2) when closing a compound consequent of an `if` statement that is followed by an `else` , and 3) when preceding the `while` part of a `do-while` statement.
4. The operators * & and − are both unary and binary.
5. : ends a label, a `case` statement, or is a separator in the ?: construction.
6. () surround the body of control statements and are used for grouping in arithmetic statements.
7. The `else if` construction is an exception to the general indentation scheme.

Using the above exceptions and the general structure of C syntax, **cb** keeps a stack for each level of curly bracket. The stack holds the current indentation level, a stack of the indentation levels of `if` statements and the depth of control statements with simple consequents. By keeping track of parentheses, curly brackets, semicolons, and a few special keywords, **cb** is able to decide where to increase and decrease the level of indentation. For example, in the fragment

```
for (i = j = 0; s[i] != ' '; i++)
        if (s[i] != c)
                s[j++] = s[i];
```

cb collects characters until it finds the blank after the `for` It identifies `for` as a keyword and notes that it is now in a control statement. It continues to collect characters, ignoring the ";" because the parenthesis are not balanced, until it encounters the ")". Since the next character is not an open bracket, it knows that the `for` has a simple consequent and increases the level of indentation by one. **cb** continues in this manner until it finds the ";" at the end of the third line. The ";" ends the consequent of the control statements and therefore causes the level of indentation to return to that of the `for`.

In the fragment

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

cb remembers the indentation levels of the `if` statements so that it can return to that level with the corresponding `else` statements.

The `do-while` statement also requires special treatment, since it is the only control statement with a complete C statement between its halves and the `while` (or preceding "}") should line up with the `do`. In the code

```
if (c == '.')
        do
                i++;
        while ((c=getchar()) != '.');
printf("%d\n",i);
```

the `while` must be indented to the same level as the `do` before the code can pop back to the indentation level of the `if`.

6. Strict Mode

In addition to adding standard indentation and enforcing one statement per line, the −s flag will cause cb to canonicalize the input to a standard layout style as follows:

1. Simple consequents of control statements are placed on a separate line and indented one tab more that the control statement, as in

```
if (a)
        b;
```

2. Open curly brackets are always preceded by a space and placed on the end on the line containing the control statement to which they belong. The matching closing bracket is alone on a line and indented to the level of the control statement, as in

```
while (a) {
        ...
}
```

3. If the body of a `do-while` statement is simple, it is on a separate line and indented one tab more than the `do`. The `while` lines up with the `do`. If the body of a `do-while` statement is compound, the `do` and the open bracket are on the same line, as are the closing curly and the `while`.
4. The argument definitions in a function definition are on separate lines and the bracket that opens the function is at the left margin.
5. Control keywords are followed by a space. Single type keywords are followed by a tab. Several type keywords in a row are separated by blanks and the last is followed by a tab.
6. Binary operators are surrounded by spaces except in subscripts.
7. Commas are followed by a space.
8. Comments, alone on a line, are indented to the same level as the next statement. Comments at the end of lines are untouched.
9. In `structure` or `union` initialization, only the first level of { increases the level of indentation.
10. Null consequents of control statements are alone and indented on the next line.
11. `else` is on the same line with the closing and opening brackets. `else if` is indented to the level of the last `if` or `else if`.
12. The bracket that ends a function is followed by two blank lines.
13. Preprocessor statements are passed through untouched.
14. Extra blank lines are preserved.

7. Line Length

There are two interacting limits on line length: first, the maximum number of leading tabs per line, and second, the maximum number of characters per line. These are separate limits to avoid the situation in which the leading white space almost equals the maximum number of characters, causing one line to be split over several that are mostly tabs. The default tab maximum is 10; the line length is 120 minus 10%, or 108. cb treats the maximum line length as a soft limit, splitting the line at the first operator or comma after the limit is reached. When the tab maximum is exceeded, cb outputs a comment and folds the code to the left margin. Succeeding code is indented from the left margin until the indent level falls below the maximum again, when another comment is output and the code returns to the proper indentation level. The user may specify the maximum line length with the flag −l *n*. The limits are then calculated as follows:

$$linelimit = n - .1*n$$

$$maxtabs = linelimit/8 - 2$$

Another line length issue is whether to join lines that were split by the user before the limit. With long arithmetic expressions the user may wish to display parallelism in the code by splitting a statement across several lines. Here it is undesirable for **cb** to put the lines back together. At the other extreme is the user who has inherited a long program in which lines have been split unnecessarily, making the code very difficult to read. By default **cb** retains all user newlines. The **-j** flag may be used to join lines in the latter case.

8. Problems

When operating in strict mode, **cb** usually produces output that is identical to the layout a person would produce. However, sometimes the decisions a person makes regarding code layout are a matter of aesthetics rather than the application of strict rules. On these occasions the **cb** output will differ slightly from that of a person. For example, although not everyone agrees that all binary operators should be surrounded by blanks, even those who agree in principle with the rule find some situations where the code is more pleasing without the blanks. Most of the aesthetic decisions made by people are made on the basis of looking at a whole expression or statement. **cb**, however, does only local beautification, seeing at most two tokens at a time. Although it might be possible for the beautifier to look at whole expressions, the payoff does not seem worth the complication of doing a more global job.

Another difference between the automatic layout and the layout done by people is in structure and array initialization. In small structures or arrays initialization frequently takes the form

```
struct date d = {4, 7, 1776, 186, "Jul"};
```

cb transforms this into

```
struct date d = {
        4, 7, 1776, 186, "Jul" };
```

Here, the beautifier is applying a strict rule and the person is using her/his knowledge that the structure has only one value.

Casts and typedefs occasionally cause **cb** to mistake a unary operator for a binary operator. **cb** attempts to solve this problem for global and external variables by assuming that occurrences of the "*" operator outside curly brackets are unary. This is almost always correct. A harder problem is presented by casts in arithmetic expressions like

```
typedef DATA int;

... (DATA)*ptr ...
```

Without knowing what DATA is, the beautifier has no way of knowing that the "*" is a unary operator.

When **cb** is run on the 2183 lines of code in [1] there are 53 lines in which the program output differs from the book layout. Most of these differences are places where the authors chose not to put blanks around binary operators. Nowhere in the program output does the addition of the blanks cause the line to grow by more than 4 or 6 characters.

9. Conclusions

A program to beautify C code has been written that operates on two different levels. At the first level it is useful to individual programmers who want to maintain their own coding style but have standard indentation automatically added to the code. At the second level, in strict mode, cb is useful to large projects that want to enforce a canonical layout style on a large body of code. The only requirement for cb to work properly is that major punctuation not be hidden in preprocessor statements. Last, but far from least, cb is useful as a debugging tool for visually pointing out missing curly brackets and `if-else` groupings.

L. Cherry

MH-1271-llc-unix
Att.
References

References

1. Brian W. Kernighan and Dennis M. Ritchie, *The C Programming Language*, Prentice-Hall Inc., Englewood Cliffs, N. J. (1978).
2. L. A. Dimino, internal memorandum, 1973.
3. Mitchell H. Clifton, "A Technique for Making Structured Programs More Readable," *SIGPLAN Notices*, 1978, vol. 13, no. 4, pp. 58-62.