C(++) Beautifier V1.6
Written By Steven De Toni December 1994

Contents

Page

1 **Introduction**

2 **Disclaimer**

3 **Requirements**

4 **How To Use The Program**

 10 **Configuration File Error Messages**

 11 **Run Time Errors During Input File Processing**

12 **C(++) Beautifier Limitations**

 14 **Contact Addresses**

Introduction

This program enables a user to reformat certain elements of a C, or C++ source code. This helps if one person's code is a little unreadable, or has been indented to a style that you dislike. Using this program will hopefully help in re-styling the code so that it closely meets your style. However, due to the many styles of C(++) that can be coded, there are limits to how well this program will handle the source code, and resulting re-formatted source.

The following are a list of features implemented:

- **Reposition comments at certain locations.**

- **Remove non-printable characters not contained within quotes.**

- **Convert non-printable characters within quotes to octal/character notation.**

- **Re-space line spacing between functions.**

- **Reposition opening braces to Kernighan/Ritchie style, or to Pascal style.**

- **Use of tabs, or spaces in indenting.**

- **Indention of single line code, according to certain C condition statements.**

<u>Disclaimer</u>

Requirements

This program will run under Microsoft Dos V3.3 and Unix platforms.

It uses approximately 50 - 70k (dependant upon internal queue size) of memory during execution.

The program code has been written in such a way as to near compatible with existing C++ compilers, however the code is not ANSI standard and may require a little rewriting. The source code has been written with standard Unix functions so that least amount of rewriting should be needed if recompiling on another computer platform.

The current sources have been compiled using Turbo C++ V1.0, GNU G++ V1.05 for MS-DOS, GNU G++ 2.6.2 under Sun SPARCstatiton running SunOS.

How To Use The Program

Operation of the program is via the command line (CLI), and with help from a configuration file you can define your format requirements. Basically each command directive starts with a dash **'-'** followed my the command name. If another parameter is need for the command, the parameter is added after the command, and separated with a space (i.e. **bcpp -fi input.cpp**).

N.B: Don't enter **bcpp.exe** on its own to find it's command help,
use **bcpp -?**, or **bcpp -h**. This is due to the input redirection from the O/S. Keeping to Unix CLI standards (if there is one), a string that is read from the CLI and does not have a command directive, it is considered a input file. If there are two such occurrence on the CLI, the precedence will be input file first, and output file second (**i.e. bcpp infile.cpp outfile.cpp -lg**), a third such occurrence will generate a error message.

If no output file is presented, it's assumed output is via
stdout, this will automatically turn off program output.
Parameters entered on the command line will override parameters
that have been read from the configuration file.

Example:

```
bcpp -fi input.cpp -f 2 -qb 5 -na -no -cc 60 > output.cpp
```

Synopsis

```
-fi input.cpp   Input file = input.cpp
-f 2            Function spacing = 2
-qb 2           Internal Queue Buffer = 5
-na             Remove non-ascii chars!
-no             Suppress program output!
-cc 60          Comments that appear on same line as
                code will be placed in column 60.
> output.cpp    Output of program is redirected to
      "output.cpp"
```

A configuration file can be used to specify most of the non-changing elements in reformatting of code, this is done via a separate file. The configuration file consists of some fairly lengthy words that are used to explain the output style of the code. However setting these parameters is very easy, they basically consist of two types, Boolean, or Integer types. Using you favourite text editor, you can change the following within the configuration file ...

The following parameters will be contained within the configuration file (default is **bcpp.cfg**). The program will attempt to read the default configuration file within the current directory each time it's executed. Using the **-fnc** option you can define a custom configuration file name, and path from the CLI.

Integer Type Ranges : 0 - 500
Boolean Type Ranges : On, Yes, or Off, No

Function_Spacing : Integer
This parameter specifies how many lines separate two functions.

E.G.

```
function_spacing      = 2
```

CLI

-f 2

Use_Tabs: Boolean
Specifies whether to use tabs in indenting code.

E.G.

```
use_tabs              = no
```

CLI

-t (This will turn tabs on, default uses spaces)
-s (Use tabs for indenting)

Indent_Spacing : Integer
Specifies how many spaces to indent. This parameter is also sets the width of tabs, bcpp considers the width of a tab to be the same as the width of an indent.

E.G.

```
    indent_spacing        = 4
```

CLI

```
        -i 4
```

Comments_With_Code : Integer
Defines the column in which comments that appear after code
on a line will be placed.

E.G.
```
comments_with_code    = 50
```

CLI
-cc 50

Comments_With_Nocode : Integer
Defines the column in which comments that appear on there one
in a line will be placed.

E.G.
```
comments_with_nocode  = 0
```

CLI
-nc 0

NonAscii_Quotes_To_Octal : Boolean
Use this option to change non-ascii (non-printable) chars to
octal notation if they lie within quotes. his parmeter doesn't take effect
unless either the Ascii_Chars_Only
or Leave_Graphic_Chars parameters have been set.

E.G.
```
NonAscii_Quotes_to_Octal = no
```

CLI
-nq (Turn off non-ascii chars in quotes to octal)
-yq (Turn on non-ascii chars in quotes to octal)

Leave_Graphic_Chars : Boolean
Setting this parameter to yes will strip non-printable characters
from the source file, but leave any characters that are IBM
graphics alone. Any non-printable characters that lie within
quotes will be transformed into octal/character notation, if
NonAscii_Quotes_To_Octal parameter is set to True.

E.G.

```
leave_graphic_chars       = yes
```

CLI

-lg

Ascii_Chars_Only : Boolean
Setting this parameter to yes will strip any non-printable, non-ascii characters from the input file. Any non-printable characters that lie within quotes will be transformed into octal/character notation if NonAscii_Quotes_To_Octal is set to True. Comment out this parameter if you are using Leave_Graphic_Chars parameter, as this parameter will override it.

```
   E.G.
       ascii_chars_only         = yes

       CLI
              -na    (Dont remove non-ascii characters)
              -ya    (Yes remove non-ascii characters)
```

Place_Brace_On_New_Line : Boolean
When set to 'on' bcpp will place opening braces on new lines ("Pascal" style C coding), when set to 'off' bcpp will use "K&R" style C coding.

```
Pascal style C coding:          if (condition)
                                {
                                   statements;
                                }

K&R style C coding:             if (condition) {
                                    statements;
                                }
```

```
   E.G.
       place_brace_on_new_line  = on

       CLI
              -bnl (on )
              -bcl (off)
```

Program_Output : Boolean
This parameter will stop output from the program corrupting output that may exit from the program via the standard output.
If this parameter is set to off/no then no output is generated from the program, unless an error is encountered. sderr is sed to display any errors encounted while processing.

E.G
```
        program_output          = off
```

CLI
-no (default is generate output if possible, this will force output off)
-yo (turn on program output if possible)

Queue_Buffer : Integer
Specifies what the internal memory requires will be in size of the line processing buffer. This essentially used only for open brace relocation in kernighan/ritchie style. Extending this buffer to large amounts of memory will slow processing!

E.G
```
        Queue_Buffer            = 2
```

CLI
-qb 2

; : Not Applicable
Placing a semi-colon in front of text makes everything after the semi-colon a comment.

Backup_File : Boolean
This option will backup the input file to a file with the extension ".bac" and overwrite the input file with the reformatted version.

E.G
```
        backup_file             = yes
```

CLI
-yb (yes, backup input file if possible)
-nb (no, don't backup input file)

Loading Configuration File : CLI only
I have a implemented a configuration setting to allow custom file selection from a specific path/file defined by a user.

E.G

```
bcpp input.cpp     -yb (attempt to read bcpp.cfg default
                       configuration file first before
        processing CLI options)

bcpp -fnc /bin/bcpp.cfg (load configuration file at
                                   said location)
```

CLI

-fnc (use user defined)

Input File Name : CLI only
This option will attempt to read data at a given path, and file name.

E.G

```
bcpp -fi input.cpp > output.cpp
```

CLI

-fi

Output File Name : CLI only
This defines the output file name that data is to be written to.

E.G

Has to be like this, (in DOS, at least):

```
bcpp  -fo output.cpp < input.cpp
```

ClI

-fo

Online Help : CLI only
Some online help which is brief but to the point exists within the program.
The help lists all of the CLI commands available within the program.

E.G bcpp -h

```
CLI bcpp -?
  bcpp -h
```

Configuration File Error Messages

If you enter a command/parameter incorrectly within the configuration file, upon the executable program reading it, the program will generate a error message along with its line number. The following is an explanation of error messages that may occur while reading parameters within the configuration file.

- **Syntax Error After Key Word** :
  Error occurs because the character/word after a parameter was incorrect, or expected another keyword (e.g =, Yes, No, On, Off)

- **Range Error** :
  Error occurs when integer parameters have a invalid numeric setting (i.e A number is not within 0 - 500).

- **Expected Numeric Data** :
  This error occurs when alpha-numeric data is in place of numeric data for integer type parameters.

- **Can't Decipher** :
  The parameter at said line is not valid (i.e not recognisable).

If any errors have occurred after reading the configuration file; the user is prompted with a [y/n] continuation prompt to either fix the configuration error(s) before processing, or continue with current set parameters.

Run Time Errors During Input File Processing

- **Memory Allocation Failed** :
The program was unable to allocate memory to process data.
This error will stop processing of data.

- **Error In Line Construction**
- **Expected Some Sort Of Code ! Data Type = ?** :
This error is generated within the line construction
process. The decoded line from the input file may be to
undecipherable for this program. Find line in input file,
and see if it can be alter so that processing can continue.

<u>C(++) Beautifier Limitations</u>

Hopefully this section will high light certain areas within code where this program will fail to reconstruct the output code to the desired style (although it may still be able to compile).

- All code that is feed through this program should be within a compilable state. This means that there should be closing braces that cancel out opening braces. This program does no syntax checking at all upon the code, but reformats it according to open, closing braces, and a hand full of key words for single line indentation.

    - Some define statements may corrupt output due to the characters they may contain (i.e '{', or '}'), or my be involved in a misplaced open brace.

- There also exists an limitation on how far the movement of open braces can be processed. This is due to the current design of the program (**this can fixed easily by extending the internal queue buffer size**), memory requirements, processing speed. Dynamic memory allocation is used extensively throughout the program, and may exceed current limits if certain conditions arise.

The example show that the movements of the brace from the new line to the above code line will not take place as it would be out of scope for the program if the internal queue buffer is limited to 2 lines in size.

Example of brace movement scope:

```
if (a == b)
// Brace will not be re-positioned
{
        b = c;
}

if (a == b)    // Brace will be re-positioned
{
        b = c;
}
```

End Result

```
if (a == b)
// Brace will not be re-positioned
{
        b = c;
}

if (a == b){   // Brace will be re-positioned
        b = c;
}
```

- There is a constraint that a single line of code should only have one type of comment. If there are both C, and C++ existing on the same line then the line construction phase of the program will become confused, and generate a error message. The following line will produce a Line Construction Error message.

Example of multiple comments.

```
/* C Comment */ a = b; // C++ Comment
```

The above line will generate an error. Remedy this by removing one type of comment, combine them, or place one on a new line.

Contact Addresses

You can contact me via various online networks:

Internet Address
tge@midland.co.nz

Net Mail Via Fido-Net (Dog Net)
Steven De Toni,
"The Great Escape",
Hamilton,
New Zealand

Demi-Monde New Zealand National Mail Net Work
(see Dog Net)

All else fails, send me snail mail at:

17 Garden Heights Ave,
Melville,
Hamilton,
New Zealand

Special thanks goes out to Glyn Webster for proof reading my manual, and testing my program.

All _grammatical_ errors within this document are there for your enjoyment. ;-)