# Testwell CMT++

## Complexity Measurement Tool for C/C++/C#

## User's Guide
## Version 6.0

# CMT++ - Complexity Measurement Tool for C / C++ / C# User's Guide

When written this document reflects the CMT++ product version 6.0

August 2015 (document upgrade v5.0 —> v6.0)
June 2012 (document upgrade v4.2 --> v5.0)
September 2007 (document upgrade v4.1 --> v4.2)
December 2005 (document upgrade v4.0 --> v4.1)
January 2005 (document upgraded v3.5 --> v4.0)
December 2003 (document upgraded v3.4 --> v3.5)
August 2002 (document upgraded v3.3 --> v3.4)
February 2002 (document upgrade v3.2 --> v3.3)
May 2001 (document upgrade v3.1 --> v3.2)
October 1999 (document upgrade v3.0 --> v3.1)



**Verifysoft Technology GmbH**
Technologiepark Offenburg
In der Spoeck 10-12
D-77676 Offenburg, Germany
URL. http://www.verifysoft.com

# Contents

# 1. About This Guide

## 1.1. Overall

This guide is written for the Testwell CMT++, Complexity Measurement Tool for C/C++/C#, version 6.0. Hereinafter called just shortly as CMT++.

CMT++ is available on many platforms including Windows and several Unix environments. This guide is intended to be used in all of those environments and describes the basic functionality and command line based use of CMT++.

On Windows platform CMT++ can be used via a graphical user interface (GUI). Effectively it is a graphical program layer for using the basic command line mode of CMT++ and for easily viewing the generated CMT++ reports. The CMT++ GUI has its own on-line help and it is not described in this guide.

The examples of this guide have been worked up at the command line prompt of a Windows machine. If your environment is something else, you should have no problems in understanding the examples and in transforming them to your environment, because effectively the only differences are in the syntax of file names.

This guide is organized as follows:

- Chapter ”2. Introducing CMT++” describes the properties and purpose of the system.
- Chapter “3. Installing CMT++” describes the overall arrangements of all CMT++ installations.
- Chapter “4. Configuring CMT++” describes how you can configure CMT++ and set ‘company standards’.

- Chapter “5. Using CMT++” gives the operating instructions of CMT++ for getting the various report types (text, XML, Excel, HTML).
- Chapter “6. Interpreting Complexity Measures” discusses the interpretation and use of the software metrics calculated by CMT++.
- "Appendix A. The Source Code Language" describes the way CMT++ analyzes C/C++/C# source code.
- "Appendix B. How the Measures Are Calculated" specifies how the individual complexity measures are calculated from the source code.
- "Appendix C. Measuring Assembly Code" describes how assembly code is measured.
- "Appendix D. cmt Error Messages" tells the cmt tool error messages.
- “Appendix E. cmt2html Error Messages“ tells the cmt2html tool error messages.
- Finally there is "Index".

## 1.2. About This Version of CMT++

Testwell CMT++, Complexity Measurement Tool for C/C++/C#, Version 6.0, follows the version v5.0 (from June 2012). The essentian enhancement in this version is better support on C# language. Already some previous versios could measure C# code, but some C# language constructs remained unindentified or caused biased measures.

At Windows platform the CMT++/Microsoft Visual Studio xxxx Integration Kit components were removed (on various VSxxxx versions). Effectively same functionality can be obtained by using straight away the CMT++ GUI.

File VERSION.TXT on the delivery media describes the more detailed history of changes in successive versions of CMT++. The initial CMT++ version was developed in 1990.

# 2. Introducing CMT++

## 2.1. About CMT++ and Complexity Metrics

Testwell CMT++ - Complexity Measurement Tool for C/C++/C# , is a tool for analyzing the static complexity and size properties of code written in C, C++ or C#. Also assembly code, either inlined in a C/C++ file or from a separate file, can be measured.

The code complexity is known to correlate with the defect rate and robustness of the application program as illustrated in the figure below:

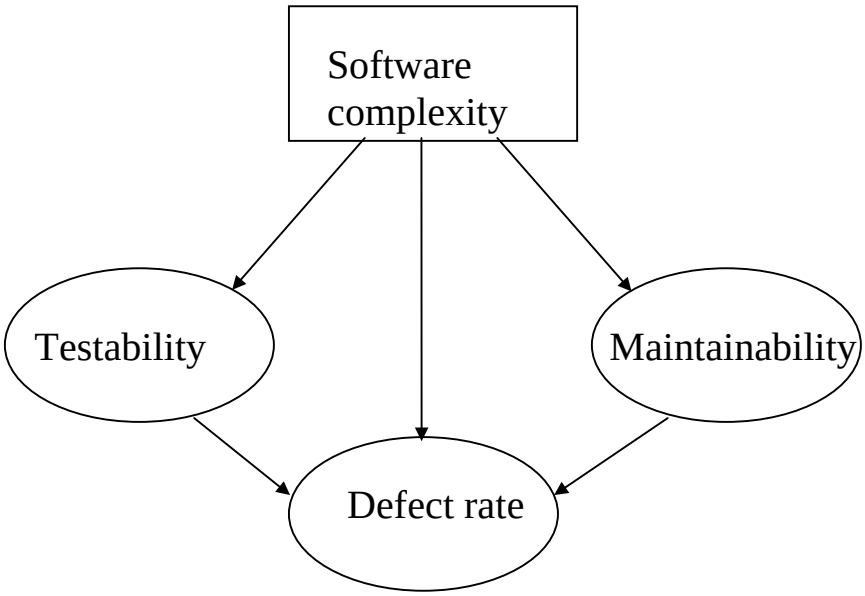



Figure: How software complexity affects quality attributes and testing.

The figure emphasizes firstly that complex code is difficult to test. And when it is difficult to test, probably more errors remain unrevealed in the final program. Secondly, complex code will be more error-prone in itself and affect the defect rate of the final program. And thirdly, complex code is difficult to maintain. And

being so, again, it is likely that more errors find their way to the final program.

There are also cost aspects here, because the testing and maintenance are major sources of the costs in software projects, too. The costs of bad quality and erroneous programs can be very high, sometimes crucial to a company. Some of these costs can be attributed to unnecessarily complex code.

Now, the question is: Do we have any means to locate the complex code so that we could avoid these risks.

CMT++ is a tool, which can be used for measuring the complexity of C, C++ and C# code. The measures include McCabe's cyclomatic number, various lines-of-code metrics, Halstead's metrics and Maintainability Index (MI).

These measures can be used in assessing the quality of the code. Based on the static properties of the program code, CMT++ gives estimates how error prone the program source code is due to its complexity, how long it will take to understand the code, what is the logical volume of the code, etc. The project team usually has not time to inspect all the code produced by the project. CMT++ can assist in locating the modules, which are most likely to cause problems in the future.

The oldest way to estimate the complexity of a program is to count the number of source lines. However, this measure depends on the formatting of the code, on the programming language, on the programming style, and is insensitive to the actual logic of the program.

Several more specific and accurate measures have been developed:

- the number of actual program lines (LOCpro), where pure comments and blank lines are ignored
- the cyclomatic number (v(G)), which measures the number of conditional branches in the flow of control
- the program volume (V), which is a measure of the information contents of a program
- estimate of the number of programming errors (B)

- estimate of the time needed to implement or understand the program code (T)

No measure as such is a magic number that can distinguish good programs from bad ones. But a set of different characteristic values can be used for filtering out potential candidates for further inspection.

### 2.2. Measures Calculated by CMT++

CMT++ reads in a set of C, C++ or C# source files and calculates the following software measures for them:

- McCabe's cyclomatic number
- Lines-of-code metrics
- Number of semicolons
- Maximum nesting depth of {} in functions
- Number of function parameters
- Halstead's metrics
- Maintainability index

McCabe's cyclomatic number is a single quantity:

**v(G)** The cyclomatic number. This measure estimates the control flow complexity of the code.

v(G) is reported per functions, per files and per over all files.

Lines-of-code metrics include four quantities:

**LOCbl** The number of blank lines.

**LOCcom** The number lines with comments. (These lines may also contain program code.)

**LOCphy** The number of physical lines.

**LOCpro** The number of lines with program code. (These lines may also contain comments.)

The above lines-of-code measures are reported per functions, per files and per over all files.

Number of semicolons is calculated over all files that are measured in one CMT++ run. The semicolons that are in string or character literals or in comment text are excluded from the figure.

Maximum nesting depth of {}s (**MaxND**) is calculated on each function. It is somewhat related to the algorithmic complexity v(G) and is an indication on how deep the algorithmic nesting structure is in a function. MaxMD is reported per functions and per files.

Number of parameters is calculated and reported per functions.

Halstead's metrics consists of the following quantities:

**B** Number of delivered bugs. This quantity is an estimate of the number of bugs in the program.

**D** Difficulty level, error-proneness.

**E** Effort to implement.

**L** Program level. This quantity represents the abstraction level of the program.

**N** Program length.

**N1** Number of operators.

**N2** Number of operands.

**n** Vocabulary size or number of unique operators and unique operands.

**n1** Number of unique operators.

**n2** Number of unique operands.

**T** Implementation time (or time to understand).

**V** Program volume or information contents of the program.

The above Halstead measured are reported per functions and per files.

The lines-of-code, McCabe and Halstead measures are further calculated with certain formulae to a MI (Maintainability Index) measure.

**MI** Maintainability Index.

**MIwoc** Maintainability Index without comments.

**MIcw** Maintainability Index comment weight.

The above MI measures are reported per functions, per files and per over all files.

In the normal course of work, CMT++ is used to calculate the above measures from the non-preprocessed source files (when C or C++ code). They are the work results of the programmers and they are the ones that need to be maintained.

C-preprocessed source files can also be fed to CMT++. However, note that "flattening" the #include files, conditional compilation and resolving of macros may cause the input file to be quite different to the non-preprocessed file.

The actual formulas for calculating the measures are presented in "Appendix B. How the Measures Are Calculated".

## 2.3. CMT++ Tool Components

*cmt:* The "basic CMT++ tool", which reads the input source files and produces a textual report of them.

*cmt2html:* A Perl utility for converting a textual CMT++ report into HTML form.

*cmtui:* (On Windows only) The CMT++ GUI for using the CMT++ tool component graphically.

# 3. Installing CMT++

On Windows platforms the installation is performed by InstallShield. On Unix platforms the installation is performed by a *makefile*, which you can edit to meet your needs and then execute for copying the tool files to their correct places.

One the delivered files is README.TXT or INSTALL.TXT. You should view it before starting the actual installation. It gives you the necessary instructions for performing the actual installation on your platform. Depending on the platform 3-9 MB free disk space is needed.

You need not have a C/C++/C# compiler available in the environment where you use CMT++.If there are any platform-specific special usage notes, you can find then from the README.TXT file from the installation directory.

# 4. Configuring CMT++

CMT++'s behavior can be tuned by modifying the configuration parameters in the configuration file cmt.ini with any text editor. The configuration file contains the CMT++ alarm limit parameters and some other tool behavior settings.

The definition of a configuration parameter has the following syntax:

PARAMETER-NAME=PARAMETER-VALUE

There must not be spaces around the equal sign. A line whose first non-whitespace character is '#' is considered a comment. Empty lines can be used.

Parameter names are case-sensitive. Some of the parameters are used for the software license identification and must not be modified by the user.

You can use environment variables in a configuration file. A construct '$(ENV_VAR_NAME)' is replaced with the value of the corresponding environment variable (with empty, if the environment variable is not known).

CMT++ searches and reads configuration files from a number of places. See section "5.2. Starting cmt from the Command Line" for a description of the command line parameter **-c** (configuration) and for the searched locations. A configuration parameter definition read later (from the same file or from another file) overrides any previously read definition.

The configuration parameters are described below.

### 4.1. Measure Alarm Limit Parameters

Most configuration parameters are used to define the acceptable limits of the complexity measures.

The limits are defined by a pair of integer valued configuration parameters. For example, the configuration parameters V_FILE_MIN and V_FILE_MAX define the lowest and highest acceptable values for source file volume (V). If volume of some file is out of these limits, CMT++ marks it with a minus sign, ‘-’, in the complexity measures report (short form). In the html representation of the report the alarms are additionally highlighted with red color.

The configuration file in the installation disk contains default settings for the measure limit parameters but you can modify them for project specific needs. The limit parameters are listed below. The limits are discussed further in the chapter "6. Interpreting Complexity Measures".

COMMENT_RATIO_FILE MIN and
COMMENT_RATIO_FILE_MAX

> Specify the lowest and highest acceptable values for the ratio of comment lines and total source code lines for a whole file. The value is a percentage number (100 * LOCcom / LOCphy), no decimals.

COMMENT_RATIO_FUNCTION_MIN and
COMMENT_RATIO_FUNCTION_MAX

> Specify the lowest and highest acceptable values for the ratio of comment lines and total source code lines for a single function. The value is a percentage number (100 * LOCcom / LOCphy), no decimals.

COMMENT_FUNCTION_MIN

> Specify the absolute minimum number of comment lines that a function should have. If the convention is to use comment a block before the actual function code, the definition NOTICE_LEADING_COMMENTS=1 should also be used.

NOTICE_LEADING_COMMENTS

Specify whether the comment lines from a comment block immediately preceding a function definition should be counted and associated to function lines, and also to its comment lines. Value 1 means yes, value 0 means no. This algorithm is a bit heuristic. The preceding comment block is recognized and associated to the function that follows it, if there are no other code lines between the comment block and the beginning of the function. After the comment block there can be zero or more empty lines. If there is an intermediate empty line (that is not enclosed in /* ... */ commenting) between the comment block, it breaks the comment block counting and association to the following function. There can be both /* ...*/ comments and // comments in the comment block.

NO_COMMENT_WARNINGS_BELOW

Specify, in LOCpro, the size of entity (function or file) of whose commenting ratio CMT++ does not give any warnings. For example, sometimes, small only a few line functions are self-evident and have no comments at all or they may have a multiline comment block causing CMT++ to complain of the commenting ratio (LOCcom/LOCphy).

LOC_FUNCTION_MIN and LOC_FUNCTION_MAX

Specify the lowest and highest acceptable number of executable lines (LOCpro) per function.

DESTRUCTOR_LOC_MIN

Specify the lowest acceptable number of LOCpro per a destructor function. In many cases, especially if the code has been initially generated by a tool, destructor functions are quite small. With this definition you can give on destructor functions a lower acceptable LOCpro limit than LOC_FUNCTION_MIN would give, and you do not get alarms of them in the complexity measures report.

LOC_FILE_MIN and LOC_FILE_MAX

Specify the lowest and highest acceptable number of executable lines (LOCpro) per source file.

B_FILE_MIN and B_FILE_MAX

Specify the lowest and highest acceptable number of errors per source file. "Error" means here the value of the complexity measure estimating the error proneness of a file (B).

B_CORRECTION_FACTOR

Argument is a positive decimal number, default 1.0. The B value (estimated number of bugs) is first calculated as it is defined by Halstead. Then, before displaying the B value to the reports and comparing it against the B-limits, it is multiplied with the given correction factor.

Some “industry practioners” consider that the original Halstead B measure should be adjusted to get better estimative value to it. This configuration parameter is CMT++’s way to do such adjustment.

V_FUNCTION_MIN and V_FUNCTION_MAX

Specify the lowest and highest acceptable volume (V) of a single function.

DESTRUCTOR_V_MIN

Specify the lowest acceptable volume (V) of a single destructor function. On destructor function this definition overrides V_FUNCTION_MIN. Otherwise the rationale with this definition is similar as with LOC_FUNCTION_MIN and DESTRUCTOR_LOC_MIN.

V_FILE_MIN and V_FILE_MAX

Specify the lowest and highest acceptable volume (V) of a source file.

MCCABE_FUNCTION_MIN and MCCABE_FUNCTION_MAX

Specify the lowest and highest acceptable values for McCabe's cyclomatic number (v(G)) per function.

MCCABE_FILE_MIN and MCCABE_FILE_MAX

Specify the lowest and highest acceptable values for McCabe's cyclomatic number (v(G)) per source file.

MCCABE_PREFERENCE

Specify in what “flavor” the MCCABE v(G) is calculated. The possible argument values are:

- *basic*: Operators &&, || are not counted to v(G).
- *extended*: Operators operator &&, || are counted to v(G). Initially, when CMT++ is installed, this setting is made active.
- *basic_modified*: Like *basic*, but ‘case n’ labels are not counted to v(G) while ‘switch()’ gives +1 to v(G).
- *extended_modified*: Like *extended*, but each ‘case n’ gives +1 to v(G) and ‘switch()’ has no effect to v(G).

MI_PREFERENCE

The possible argument values are MI or MIwoc. The argument specifies:

- which one, Maintainability Index [with comments] (MI) or Maintainability Index without comments (MIwoc) is shown in the default short form textual CMT++ report,
- and do the MI_FUNCTION_MIN and MI_FILE_MIN limit values apply on the MI or MIwoc measures correspondingly.

MI_FILE_MIN

Specify the lowest acceptable Maintainability Index (MI or MIwoc) value of a single file.

MI_FUNCTION_MIN

Specify the lowest acceptable Maintainability Index (MI or MIwoc) value of a single function.

### 4.2. Excel Field Separator

With the **-x** option CMT++ produces its output to a text file in a form, which is suitable for Excel (or a similar spreadsheet utility) input file. With setting

EXCEL_FIELD_SEPARATOR

Specify the ASCII value of the character, which will be used as a field separator in the file. For example, the value 9 means tab character ('\t') and 59 means a semicolon (';').

### 4.3. C# Code Handling Parameters

CSHARP_FILE_EXTENSIONS

Initially CMT++ was designed to measure C and C++ code, where C was considered to be subset of C++. Support on C# was added later. Because C# has some special language constructs (compared to C++), for correct parsing, CMT++ needs to know if the input is C/C++ or C#.

If the input file has one of the extensions listed in this setting, CMT++ takes the code to be C#, Normally this setting has only CSHARP_FILE_EXTENSIONS=cs.

### 4.4. Assembly Code Handling Parameters

ASSEMBLY_FILE_EXTENSIONS

Specify the file extensions, which CMT++ should consider to be assembly files and measure as assembly code. For example, with ASSEMBLY_FILE_EXTENSIONS=asm,s,as setting the files, whose names end on ".asm", ".s" and ".as"

are measured as assembly files. Use no spaces in the argument.

ASSEMBLY_ID_ADDON_CHARACTERS

When parsing assembly code, either from a totally separate assembly file or from a C/C++ file (inlined assembly), the specified characters are additionally associated to belong into assembly identifiers. For example, the setting might be ASSEMBLY_ID_ADDON_CHARACTERS=.@ . Corollary, when parsing assembly code, identifiers like ".abc.def@123" and "__AZaz19$.@" are considered as one assembler identifier. When CMT++ knows it parses C/C++ code, the characters associated into one identifier can be composed of A-Z, a-z, 1-9, _ and $.

ASSEMBLY_COMMENT_CHARACTER

When parsing assembly code, either from a totally separate assembly file or from a C/C++ file (inlined assembly), the specified character is considered to start assembly comment (line-comment, to the end of the line). For example, the following might be a reasonable definition for many assemblers: ASSEMBLY_COMMENT_CHARACTER=; .

### 4.5. Software Licence Parameters

The configuration parameters TOOL, USER, COMPUTER, LICENCE, EXPIRATION, NOTE1, NOTE2, NOTE3, NOTE4, NOTE5, TARGET_CHECK and CONTROL are used for identifying the CMT++ software license. Do not modify their settings. If they are modified in an unauthorized way, CMT++ will not work any more.

### 4.6. Hardware Control Key Port

This parameter is needed only in environments (PC), where a hardware control key, which is plugged into a parallel port, controls the license. The port number is the number of the user

selectable LPT port where the control key is attached to. For example, the following definition is correct.

KEYPORT=1

Remark: Hardware control key (dongle) based license control is no more in use. This parameter is a relic of some old versions, where it still was supported.

### 4.7. Link to Floating License Manager

In some environments the license may be controlled by FLEXlm license manager. The license may be a floating license or a node-locked license. The parameter FLEXLM_LICENSE_FILE specifies the path and name of the testwell.lic (or testwell.dat) license file, or port and host of the license manager daemon. Examples:

FLEXLM_LICENSE_FILE=27000@flxserver

Floating license. License manager daemon runs on machine flxserver, port 27000 will be used.

FLEXLM_LICENSE_FILE=@flxserver

Like above, but FLEXlm finds the port number to use.

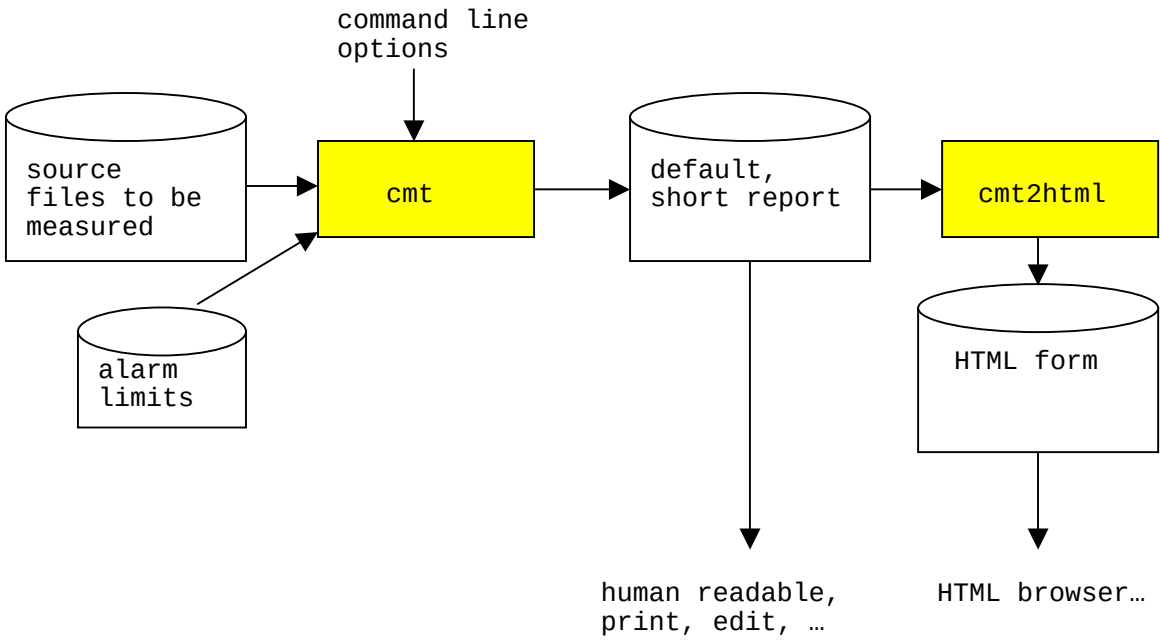
FLEXLM_LICENSE_FILE=

Perhaps a floating license, but the connection is found by environment variables LM_LICENSE_FILE or TESTWELLD_LICENSE_FILE, or based on the value of TESTWELLD_LICENSE_FILE in registry (Windows), or based on the file $(HOME)/.flexlmrc (Unix).

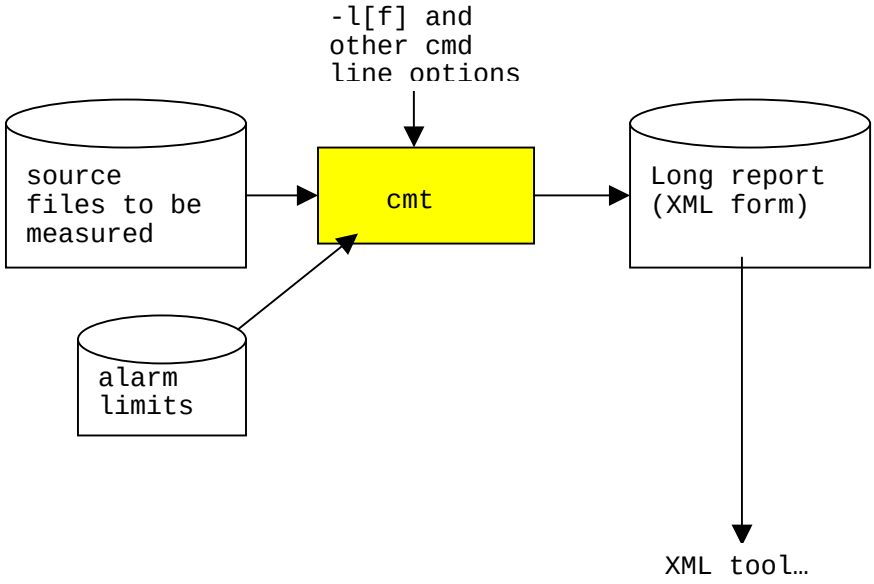
# 5. Using CMT++

## 5.1. Overall Architecture

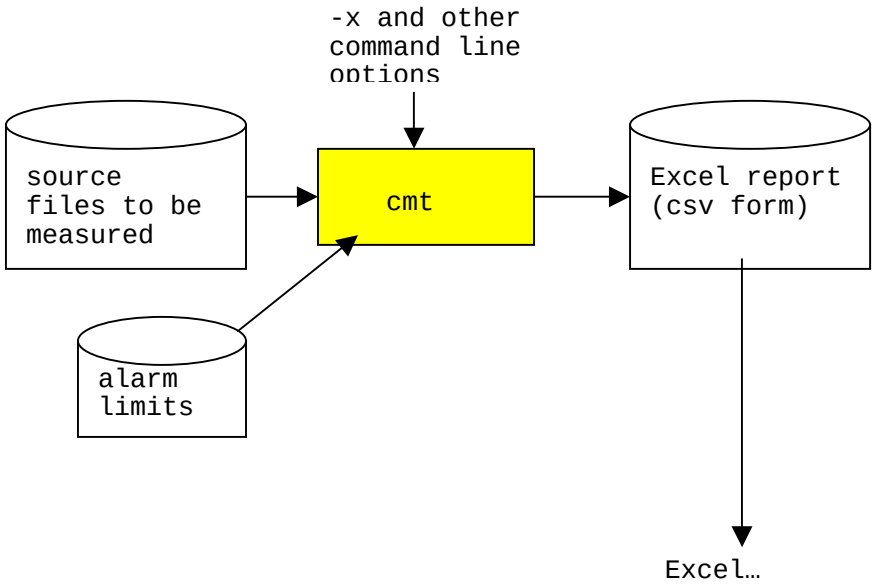
Simply, CMT++ is a tool, which analyzes a set of C, C++, C# or assembly source files and writes a textual complexity measures report of them. You can select by command-line parameters what kind of report will be written. With some add-on tools the report can be processed further into other representations.The next pictures show the major alternatives to use the "CMT++ tool chain". Usage is from command line. On Windows there is additionally a GUI to use the same.




Default, human readable reporting.




XML-reporting. Usable in "build integrations".




Excel reporting.

## 5.2. Starting cmt from the Command Line

CMT++ is started from the command line in one of the following ways:

For getting a short or a somewhat longer on-line help:

**cmt -h | -H**

For producing a short tabular form of the report (default format, textual, can be viewed with any text editor and/or worked onwards to HTML form):

**cmt** [confsettings] [**-v**] [**-s**] [**-w**] [**-f** *filenames*] [**-o** *outfile*] [*sourcefile...*]

For producing a long form of the report (XML format, textual):

**cmt** [confsettings] [**-v**] **-l**[**f**] [**-s**] [**-f** *filenames*] [**-o** *outfile*] [*sourcefile...*]

For producing an Excel input file form of the report (textual):):

**cmt** [confsettings] [**-v**] **-x** [**-nxh**] [**-s**] [**-w**] [**-f** *filenames*] [**-o** *outfile*] [*sourcefile...*]

Where the confsettings is [**-c** *conffiles*] [**-C** *confparam=value*]...

The above four alternatives to invoke CMT++ specify also the allowed combinations of the options.

The meanings of the command line options and arguments are:

**-h** ("small help) Displays a synopsis of the command line options. If **-h** option is given, the command line must not contain other arguments.

**-H** ("big help") Displays a few screens of general help about CMT++. If **-H** option is given, the command line must not contain other arguments.

**-c** *conffiles*

("additional configurations") Specifies one or more configuration files to be read by CMT++ after to the default

locations for configuration files have been searched. If more than one file is given, they must be separated by a semicolon. There must be one space after the **-c** option but, if multiple configuration files are specified, no spaces around the ';'s. An example:
-c e:\grpdir\grpsettings.ini;e:\prjdir\prjdir\prjsettings.ini.

The configuration files are searched from the following places in order (a later definition overrides a previous one):

1) File */usr/local/lib/cmt/cmt/cmt.ini* (Unix only).
2) File *$HOME/lib/cmt/cmt.ini* (Unix only).
3) File CMT.INI (in Unix *cmt.ini*) in the directory specified by the environment variable CMTHOME.
4) file *.cmt.ini* in the user's home directory (Unix only).
5) file specified by the environment variable CMTINIT. Multiple files can be specified, separated by a semicolon.
6) File *CMT.INI* in the current directory (file *.cmt.ini* in Unix).
7) File(s) explicitly specified by the **-c** option.

After the configuration files are loaded there still may come explicit configuration parameter overridings with **-C** command-line parameter.

**-C** *confparam=value*

Override value of a configuration parameter (other than license control parameter). There must be one space after **-C** but no other spaces. There can be many **-C** options on the command line.

**-v** ("verbose") Show on the screen what configuration files CMT++ tried to find and loaded if the file was found.

**-s** ("summary") File level summary only. In the absence of this option the output file contains measures both of the file internal functions, classes and structs and of the file summary level . (If the **-x** option is also present, the behavior is slightly different, see below)

**-w** ("warnings only") Only lines that have a warning flag '-' are written to the output file. Usable when producing the short tabular form report or Excel data file. Can not be used with long report.

**-x** ("excel output") The output file produced is of the Excel input data file format. Normally, the **-s** option is not present together with this **-x** option. In such a case the resultant Excel data file contains measures of the functions in the file, not any file summary level data. If both the **-x** and **-s** options are present, the Excel data file contains only file summary level data, not any function level data.

**-nxh** ("no excel header") When producing the Excel data file output, no column headers are written to the first line of the file, only the actual data lines.

**-l** ("long") CMT++ produces the complexity measures report in a long format, in XML format, where all measures are included.

**-lf** ("long with frequencies") CMT++ produces the complexity measures report in the long format as with the **-l** option but includes also operator and operand frequencies in the report.

**-f** *filenames*

("file names") Specifies a text file that has the names of the files to be measured, one file name on a line. #-starting lines can be used as comments. Empty line or end-of-file ends the file names list. If the **-f** option is present, there no more can be source file names on the command line.

**-o** *outfile*

("output file") Specifies the output file. If the **-o** option is present, it must be followed by a filename separated by a space. The complexity measures report is then written to that file (and silently overwriting the possible previous file with that name). In the absence of this option the report is written to *stdout*.

*sourcefile...*

A list of the file names (separated by a space) to be measured. Wildcards can be used. Character '-' means the file *stdin*, i.e. the file to be measured is read from *stdin*. If no source files are given, CMT++ prompts for the file names interactively.

For example, the command

**cmt *.c**

tells CMT++ to analyze all files in the working directory that have the extension .c. The report is written in tabular format to *stdout*. Whereas the command

**cmt -c johnscmt.ini -lf -o report.xml *.c reuse\fi*.cpp**

tells CMT++ to

- analyze all C source files in the working directory and the files whose names start with fi and whose extension is .cpp in the subdirectory reuse.
- use the additional configuration settings in the file johnscmt.ini.
- produce the report as long report (XML format) to the file report.xml.
- report all possible measurements (long listing, operand frequencies included).

## 5.3. Using cmt Interactively

If no source files are given on the command line, CMT++ prompts them interactively from *stdin*. For example

```
C:\TESTDIR> cmt -o report.txt
*****************************************************************
*  CMT++, Complexity Measurement Tool for C/C++/C#, Version 6.0   *
*                                                                 *
*               Copyright (c) 1993-2013 Testwell Oy               *
*       Copyright (c) 2013-2015 Verifysoft Technology GmbH        *
*****************************************************************

Invoke the tool with option -h for help about CMT++.
Type '?' in a prompt for context sensitive help.
Type '!' in a prompt for visiting operating system.
```

```
Names of the source code files?
SOURCE ( 1) => file1.cpp
SOURCE ( 2) => file2.cpp
SOURCE ( 3) =>

Measuring file 1 2 Done

C:\TESTDIR>
```

The SOURCE ( n) => prompt is repeated until your answer with an empty line (or EOF is met in the reading). Here two files *file1.cpp* and *file2.cpp* are measured and the results are written to the file *report.txt*. The result file is of short tabular form (neither **-x** nor **-l/-lf** options were present). As the report is written to a file (**-o** option present), the screen is "free" and the text "Measuring file 1 2 Done" progress reporting is written to the screen.

You could have got the same behavior directly from the command line with the command

```
C:\TESTDIR> cmt -o report.txt file1.cpp file2.cpp
```

or as follows

```
C:\TESTDIR> cmt file1.cpp file2.cpp > report.txt
```

or, if wildcard notation *file?.cpp* matches only to these two files, as follows

```
C:\TESTDIR> cmt -o report.txt file?.cpp
```

### 5.4. Piping Source File Names to cmt

This style of using CMT++ is actually a variant of using CMT++ interactively. The difference is that the source file names are read from a file, which is piped to *stdin*, instead of typing the file names interactively from the terminal. An example:

```
C:\TESTDIR> cmt -o report.txt < files.lst
```

which is effectively the same as

```
C:\TESTDIR> cmt -f files.lst -o report.txt
```

You have prepared a text file files.lst, which contains the names of the files to be measured, one file name at a line. CMT++ believes to be running in the interactive mode and reads the answers to the

SOURCE ( n) => prompts from the piped file. EOF (or when an empty line is met) ends the source file name reading. In the file files.lst the lines starting with the '#' character are comment lines.

Another example:

```
C:\TESTDIR> dir *.cpp /s /b | cmt -o report.txt
```

Here with the operating system command a bare file name listing of **.cpp* files is produced from the current directory including its subdirectories. The file name list is piped to CMT++, which produces the *report.txt* file.

Note that the following

```
C:\TESTDIR> dir *.cpp /s /b | cmt > report.txt
```

will **not** work smoothly, because the interactive mode tool header box and source file prompting texts get also copied to the output file.

### 5.5. Reading the Actual Source File from *stdin*

You recall that '-' as a source file name means *stdin*. This being so, the following two commands are functionally equivalent:

```
C:\SOURCEDIR> type file1.cpp | cmt -o file1report.txt -
C:\SOURCEDIR> cmt -o file1report.txt file1.cpp
```

Well, there is a slight difference: in the first form CMT++ thinks to be reading a file named "-" (CMT++'s escape notation for *stdin*) while in the latter form CMT++ correctly knows that it reads the file *file1.cpp*.

The above gives you no added functionality, but if you want to measure a preprocessed source file, you can take use of this feature. Here's an example (MSVC++):

```
C:\SOURCEDIR> cl /E file1.cpp | cmt -o prefile1.txt -
```

which is equivalent to the sequence of commands:

```
C:\SOURCEDIR> cl /E file1.cpp > tempfile.cpp
C:\SOURCEDIR> cmt -o prefile1.txt tempfile.cpp
C:\SOURCEDIR> del tempfile.cpp
```

## 5.6. Example

In the following an example of measuring the complexity of four source files is presented. The files are stack.h, stack.cpp, demofile.h and demofile.cpp.

stack.h:

```
// ------------------------------------------------------------------
//
// Class    : STACK from STACK.H
// Parents  :
// Friends  :
// Part of  :
// Created  : 12 Oct 1990 by A.C.Coder
// Abstract : This module implements a dynamic stack for integers.
//            Other stacks can be easily constructed by changing
//            the defintion of STACK_ITEM type. Note: STACK_ITEM
//            should not be any array type.
// Revision : 1.0, 12 Oct 1990 12:00:00, by ACC
//
//              Copyright (C) 1990 Sample Software Ltd.
// ------------------------------------------------------------------
// Revision history
//
// 1.0: Initial revision
//
// ------------------------------------------------------------------

#ifndef STACK_H
#define STACK_H


// ------------------------- PUBLIC TYPES -------------------------

//
// STACK_ITEM: The items stored in the stack are of this type
//

typedef int STACK_ITEM;


// ----------------------- CLASS DECLARATION ----------------------


class Stack
{
public:
    Stack();
    ~Stack();
    void clear();
    unsigned height();
    void push(
        STACK_ITEM item);
    STACK_ITEM pop();
    STACK_ITEM top();
protected:
    STACK_ITEM &element(unsigned index);
private:
    unsigned   myheight; // current height of stack
    unsigned   size;     // current size (max. height)
```

```
    STACK_ITEM *value;   // the data in stack
};


// ------------------ INLINE MEMBER DEFINTIONS -------------------

#endif
```

stack.cpp:

```
// ---------------------------------------------------------------
//
// Class    : STACK from STACK.CPP
// Parents  :
// Friends  :
// Part of  :
// Created  : 12 Oct 1990 by A.C.Coder
// Abstract : This module implements a stack.
// Revision : 1.0, 12 Oct 1990 12:00:02, by ACC
//
//              Copyright (C) 1990 Sample Software Ltd.
// ---------------------------------------------------------------
// Revision history
//
// 1.0: Initial revision
//
// ---------------------------------------------------------------

// -------------------- APPLICATION INCLUDES ----------------------

#include "stack.h"

// ----------------------- PRIVATE CONSTANTS ----------------------

//
// - SIZE_STEP: defines the number of new stack elements allocated
//   each time the stack becomes full
//

int const SIZE_STEP = 40;

// ---------------------------------------------------------------
//
// Function    : Stack::Stack
// Description : Initializes the private members, all to 0. Memory
//               is not allocated before the first push.
// Updates     : - height
//               - size
//               - value
//
// ---------------------------------------------------------------

Stack::Stack():
    myheight(0),
    size(0),
    value(0)
{
    // Nothing
}

// ---------------------------------------------------
//
// Function    : Stack::~Stack
// Description :
```

```
// Updates     : - value
//
// -------------------------------------------------

Stack::~Stack()
{
    delete value;
}

// -------------------------------------------------
//
// Function    : Stack::clear
// Description :
// Updates     : - height
//
// -------------------------------------------------

void Stack::clear()
{
    myheight = 0;
}

// -------------------------------------------------
//
// Function    : Stack::empty
// Description :
// Updates     :
//
// -------------------------------------------------

unsigned Stack::height()
{
    return myheight;
}

// -------------------------------------------------
//
// Function    : Stack::push
// Description :
// Updates     : - height
//               - size
//               - value
//
// -------------------------------------------------

void Stack::push(
    STACK_ITEM item)
{
    if (myheight >= size)
    {
        size += SIZE_STEP;
        STACK_ITEM *new_value = new STACK_ITEM[size];
        for (unsigned i = 0; i < myheight; i++)
            new_value[i] = value[i];
        delete value;
        value = new_value;
    }
    value[myheight++] = item;
}

// -------------------------------------------------
//
// Function    : Stack::pop
// Description :
// Updates     : - height
```

```
//
// ------------------------------------------------


STACK_ITEM Stack::pop()
{
    STACK_ITEM item;

    if (myheight > 0)
        item = value[--myheight];

    return item;
}

// ------------------------------------------------
//
// Function    : Stack::top
// Description :
// Updates     :
//
// ------------------------------------------------

STACK_ITEM Stack::top()
{
    STACK_ITEM item;

    if (myheight > 0)
        item = value[myheight - 1];

    return item;
}

STACK_ITEM &Stack::element(unsigned index) { return value[index]; }
```

## demofile.h:

```
// This code just demonstrates what CMT++ calculates of various
// C/C++ language constructs. This file compiles ok, but as
// executable code this is nonsense.

extern int a;

class MyClass {
   MyClass() {
      a = 5;
   }
   ~MyClass() {
      a = 0;
   }
   int foo1(int i);
   int foo2(int i);
};

int SomeFunction();
```

## demofile.cpp:

```
// This code just demonstrates what CMT++ calculates of various
// C/C++ language constructs. This file compiles ok, but as
// executable code this is nonsense.

#include "demofile.h"
```

```
int a;

int MyClass::foo1(int i) {
   if (i > 5 || i > 6 || i > 7 || i > 8 || i > 9) {
      a = a + i;
   }
   return a;
}

int MyClass::foo2(int i) {
   if (i > 10) {
      a--;
   } else {
      a = a - i;
   }
   return 0;
}

int SomeFunction() {
   if (a > 0) {
      switch (a) {
         case 0:
         case 1:
         case 2:
            a++;
            break;
         case 3:
            a--;
            break;
         default:
           a = 0;
      }
   }
   return a == 0 ? 100 : 200;
}
```

We now measure the complexity of these files:

```
cmt -o report.txt stack.h demofile.h stack.cpp demofile.cpp
**********************************************************************
*  CMT++, Complexity Measurement Tool for C/C++/C#, Version 6.0      *
*                                                                    *
*               Copyright (c) 1993-2013 Testwell Oy                  *
*       Copyright (c) 2013-2015 Verifysoft Technology GmbH           *
**********************************************************************

Measuring file 1 2 3 4 Done
```

The resulting report file *report.txt* is of a short form and is listed below:

```
**********************************************************************
*  CMT++, Complexity Measurement Tool for C/C++/C#, Version 6.0      *
*                                                                    *
*                     COMPLEXITY MEASURES REPORT                     *
*                                                                    *
*               Copyright (c) 1993-2013 Testwell Oy                  *
*       Copyright (c) 2013-2015 Verifysoft Technology GmbH           *
**********************************************************************


This report was produced at Mon Aug 10 13:46:35 2015
Options: -o report.txt
```

File: stack.h

| Line | Measured object | v(G) | LOCphy | LOCpro | c% | V | B | MI |
|---|---|---|---|---|---|---|---|---|
| 60 | stack.h | 2 | 60 | 22 | | 343 | 0.09 | 118 |

File: demofile.h

| Line | Measured object | v(G) | LOCphy | LOCpro | c% | V | B | MI |
|---|---|---|---|---|---|---|---|---|
| 8 | MyClass:: | | | | | | | |
| 8 | MyClass() | 1 | 3 | 3- | | 20- | 0.00 | 137 |
| 11 | ~MyClass() | 1 | 3 | 2 | | 20- | 0.00 | 137 |
| 18 | demofile.h | 1 | 18 | 12 | - | 156 | 0.04 | 142 |

File: stack.cpp

| Line | Measured object | v(G) | LOCphy | LOCpro | c% | V | B | MI |
|---|---|---|---|---|---|---|---|---|
| 32 | Stack:: | | | | | | | |
| 32 | Stack() | 1 | 18 | 6 | | 56 | 0.01 | 150 |
| 51 | ~Stack() | 1 | 12 | 4 | | 27 | 0.01 | 160 |
| 64 | clear() | 1 | 12 | 4 | | 33 | 0.01 | 159 |
| 77 | height() | 1 | 12 | 4 | | 29 | 0.01 | 159 |
| 90 | push() | 3 | 24 | 14 | | 288 | 0.10 | 130 |
| 115 | pop() | 2 | 18 | 7 | | 94 | 0.03 | 141 |
| 134 | top() | 2 | 17 | 7 | | 100 | 0.03 | 143 |
| 152 | element() | 1 | 1 | 1- | | 52 | 0.01 | 150 |
| 152 | stack.cpp | 5 | 152 | 49 | | 972 | 0.41 | 143 |

File: demofile.cpp

| Line | Measured object | v(G) | LOCphy | LOCpro | c% | V | B | MI |
|---|---|---|---|---|---|---|---|---|
| 9 | MyClass:: | | | | | | | |
| 9 | foo1() | 6 | 6 | 6 | | 164 | 0.04 | 114 |
| 16 | foo2() | 2 | 8 | 8 | | 113 | 0.03 | 112 |
| 25 | SomeFunction() | 5 | 17 | 17 | - | 209 | 0.07 | 96 |
| 41 | demofile.cpp | 11 | 41 | 33 | - | 620 | 0.22 | 120 |

OVERALL SUMMARY:

| | 4 Files | | | 13 Functions | | |
|---|---|---|---|---|---|---|
| Measure | Alarmed | % | Limits | Alarmed | % | Limits |
| Complexity (extended) v(G) | 0 | 0 | 1-100 | 0 | 0 | 1-10 |
| Program lines LOCpro | 0 | 0 | 4-400 | 2 | 15 | 4-40 |
| Comment % | 2 | 50 | 30-75 | 1 | 7 | 30-75 |
| Volume V | 0 | 0 | 100-8000 | 2 | 15 | 20-1000 |
| Estimated number of bugs B | 0 | 0 | 0-2 | 0 | 0 | n/a |
| Maintainability index MI | 0 | 0 | 65- | 0 | 0 | 65- |

```
========================  ==================  ==================
Total                                2   8                5  7

Files: 4 LOCphy: 271   LOCbl: 45  LOCpro: 116  LOCcom: 113  ';': 51
v(G) : 16  MI without comments: 107  MI comment weight: 40  MI: 147
```

From the header file stack.h only the file summary level measures are reported. But the other header file demofile.h contained two functions with implementation (MyClass::MyClass(){…} and MyClass::~MyClass(){…}) and they are reported in addition to the file summary measures.

File stack.cpp is rather simple and CMT++ considers it quite well built. Only one alarm was reported. Function element() was all written on one line, while CMT++ would have liked to see at least 4 lines in a function. (For a class destructor a separate low limit is used, if you are wondering why the there is no warning on MyClass::~MyClass()).

Some functions and files were alarmed for their commenting % ratio. However, functions that are very small (configurable in cmt.ini file, here 10 lines) are excused from comment % alarms.

In file stack.cpp the member function bodies are preceded with a comment block. It is configurable in the cmt.ini file if the lines of such preceding comment block is included to the LOCphy lines of the function.

In regard to McCabe's complexity the most complex function was MyClass::foo1(). McCabe measure 1 is obtained when the "code runs straight through". Each conditional branch adds 1 to McCabe.

In regard to Halstead's complexity the most complex function was Stack::push(). This short report shows V, volume or a kind logical size estimate, and B, estimated number of bugs. Halstead measures are derived from the source code when it is stripped from comments and line layout, and viewed only as a sequence of operand and operator tokens.

In regard to maintainability (MI) the poorest function was SomeFunction(). The MI measure is a balanced measure that is derived with certain formulae (defined by Sofwtare Engineering Institute) based on McCabe measure, Halstead measure, and lines

of code/commenting ratio measure. MI tries to capture into one number the maintainabililty of the piece of code (single function, single file, or the whole system (=all measured files)).

The long report (with operand and operator frequencies) could have been produced as follows:

```
cmt -lf -o lfreport.xml demofile.cpp stack.h demofile.h stack.cpp
```

The long format output file is in XML, and looks as follows:

```
<?xml version="1.0" encoding="ISO-8859-1"?>
<cmt_long_report>
      <header_info>
            <cmt_version>6.0</cmt_version>
            <copyright>Copyright (c) 1993-2013 Testwell
Oy</copyright>
            <copyright>Copyright (c) 2013-2015 Verifysoft Technology
GmbH</copyright>
            <license_notes>
            </license_notes>
            <date>Mon Aug 10 14:03:26 2015</date>
            <cmt_options> -lf -o lfreport.xml</cmt_options>
            <run_directory>F:\cmtwork\v60\doc</run_directory>
            <mccabe_preference>extended</mccabe_preference>
      </header_info>
      <file name="stack.h" stamp="1188737631">
            <file_total>
                  <vG_b>2</vG_b>
                  <vG_e>2</vG_e>
                  <vG_b_max>1</vG_b_max>
                  <vG_b_avg>1</vG_b_avg>
                  <vG_e_max>1</vG_e_max>
                  <vG_e_avg>1</vG_e_avg>
                  <params>0</params>
                  <LOCphy>60</LOCphy>
                  <LOCpro>22</LOCpro>
                  <LOCbl>12</LOCbl>
                  <LOCcom>29</LOCcom>
                  <N>68</N>
                  <N1>38</N1>
                  <N2>30</N2>
                  <n>33</n>
                  <n1>15</n1>
                  <n2>18</n2>
                  <V>343.019</V>
                  <B>0.088</B>
                  <D>12.500</D>
                  <E>4287.735</E>
                  <L>0.080</L>
                  <T>00:03:58</T>
                  <MaxND>0</MaxND>
                  <MIwoc>74</MIwoc>
                  <MIcwc>44</MIcwc>
                  <MI>118</MI>
                  <operators>
                        <token count="3">#</token>
                        <token count="1">&amp;</token>
                        <token count="8">()</token>
```

```
                  <token count="1">*</token>
                  <token count="3">:</token>
                  <token count="13">;</token>
                  <token count="1">class</token>
                  <token count="1">endif</token>
                  <token count="1">ifndef</token>
                  <token count="1">private</token>
                  <token count="1">protected</token>
                  <token count="1">public</token>
                  <token count="1">typedef</token>
                  <token count="1">{}</token>
                  <token count="1">~</token>
            </operators>
            <operands>
                  <token count="2">STACK_H</token>
                  <token count="6">STACK_ITEM</token>
                  <token count="3">Stack</token>
                  <token count="1">clear</token>
                  <token count="1">define</token>
                  <token count="1">element</token>
                  <token count="1">height</token>
                  <token count="1">index</token>
                  <token count="1">int</token>
                  <token count="1">item</token>
                  <token count="1">myheight</token>
                  <token count="1">pop</token>
                  <token count="1">push</token>
                  <token count="1">size</token>
                  <token count="1">top</token>
                  <token count="4">unsigned</token>
                  <token count="1">value</token>
                  <token count="2">void</token>
            </operands>
      </file_total>
</file>
<file name="demofile.h" stamp="1188747583">

      ... for saving space contents here deleted

</file>
<file name="stack.cpp" stamp="1188737626">

      ... for saving space contents here deleted

</file>
<file name="demofile.cpp" stamp="1188747596">
      <function name="MyClass::foo1()">
            <start_line>9</start_line>
            <vG_b>2</vG_b>
            <vG_e>6</vG_e>
            <params>1</params>
            <LOCphy>6</LOCphy>
            <LOCpro>6</LOCpro>
            <LOCbl>0</LOCbl>
            <LOCcom>0</LOCcom>
            <N>38</N>
            <N1>19</N1>
            <N2>19</N2>
            <n>20</n>
            <n1>10</n1>
            <n2>10</n2>
            <V>164.233</V>
            <B>0.045</B>
            <D>9.500</D>
            <E>1560.216</E>
```

```
            <L>0.105</L>
            <T>00:01:26</T>
            <MaxND>2</MaxND>
            <MIwoc>114</MIwoc>
            <MIcwc>0</MIcwc>
            <MI>114</MI>
            <operators>
                  <token count="1">()</token>
                  <token count="1">+</token>
                  <token count="1">::</token>
                  <token count="2">;</token>
                  <token count="1">=</token>
                  <token count="5">&gt;</token>
                  <token count="1">if()</token>
                  <token count="1">return</token>
                  <token count="2">{}</token>
                  <token count="4">||</token>
            </operators>
            <operands>
                  <token count="1">5</token>
                  <token count="1">6</token>
                  <token count="1">7</token>
                  <token count="1">8</token>
                  <token count="1">9</token>
                  <token count="1">MyClass</token>
                  <token count="3">a</token>
                  <token count="1">foo1</token>
                  <token count="7">i</token>
                  <token count="2">int</token>
            </operands>
      </function>
      <function name="MyClass::foo2()">
            <start_line>16</start_line>
            <vG_b>2</vG_b>
            <vG_e>2</vG_e>
            <params>1</params>
            <LOCphy>8</LOCphy>
            <LOCpro>8</LOCpro>
            <LOCbl>0</LOCbl>
            <LOCcom>0</LOCcom>
            <N>27</N>
            <N1>15</N1>
            <N2>12</N2>
            <n>18</n>
            <n1>11</n1>
            <n2>7</n2>
            <V>112.588</V>
            <B>0.035</B>
            <D>9.429</D>
            <E>1061.544</E>
            <L>0.106</L>
            <T>00:00:58</T>
            <MaxND>2</MaxND>
            <MIwoc>112</MIwoc>
            <MIcwc>0</MIcwc>
            <MI>112</MI>
            <operators>
                  <token count="1">()</token>
                  <token count="1">-</token>
                  <token count="1">--</token>
                  <token count="1">::</token>
                  <token count="3">;</token>
                  <token count="1">=</token>
                  <token count="1">&gt;</token>
                  <token count="1">else</token>
```

```
                <token count="1">if()</token>
                <token count="1">return</token>
                <token count="3">{}</token>
            </operators>
            <operands>
                <token count="1">0</token>
                <token count="1">10</token>
                <token count="1">MyClass</token>
                <token count="3">a</token>
                <token count="1">foo2</token>
                <token count="3">i</token>
                <token count="2">int</token>
            </operands>
        </function>
        <function name="SomeFunction()">
            <start_line>25</start_line>
            <vG_b>5</vG_b>
            <vG_e>5</vG_e>
            <params>0</params>
            <LOCphy>17</LOCphy>
            <LOCpro>17</LOCpro>
            <LOCbl>0</LOCbl>
            <LOCcom alarmed="1">0</LOCcom>
            <N>45</N>
            <N1>28</N1>
            <N2>17</N2>
            <n>25</n>
            <n1>16</n1>
            <n2>9</n2>
            <V>208.974</V>
            <B>0.072</B>
            <D>15.111</D>
            <E>3157.822</E>
            <L>0.066</L>
            <T>00:02:55</T>
            <MaxND>3</MaxND>
            <MIwoc>96</MIwoc>
            <MIcwc>0</MIcwc>
            <MI>96</MI>
            <operators>
                <token count="1">()</token>
                <token count="1">++</token>
                <token count="1">--</token>
                <token count="2">:</token>
                <token count="6">;</token>
                <token count="1">=</token>
                <token count="1">==</token>
                <token count="1">&amp;gt;</token>
                <token count="1">?</token>
                <token count="2">break</token>
                <token count="4">case ...:</token>
                <token count="1">default</token>
                <token count="1">if()</token>
                <token count="1">return</token>
                <token count="1">switch()</token>
                <token count="3">{}</token>
            </operators>
            <operands>
                <token count="4">0</token>
                <token count="1">1</token>
                <token count="1">100</token>
                <token count="1">2</token>
                <token count="1">200</token>
                <token count="1">3</token>
                <token count="1">SomeFunction</token>
```

```
                <token count="6">a</token>
                <token count="1">int</token>
        </operands>
</function>
<file_total>
        <vG_b>7</vG_b>
        <vG_e>11</vG_e>
        <vG_b_max>5</vG_b_max>
        <vG_b_avg>3</vG_b_avg>
        <vG_e_max>6</vG_e_max>
        <vG_e_avg>4</vG_e_avg>
        <params>2</params>
        <LOCphy>41</LOCphy>
        <LOCpro>33</LOCpro>
        <LOCbl>5</LOCbl>
        <LOCcom alarmed="1">3</LOCcom>
        <N>115</N>
        <N1>64</N1>
        <N2>51</N2>
        <n>42</n>
        <n1>22</n1>
        <n2>20</n2>
        <V>620.117</V>
        <B>0.224</B>
        <D>28.050</D>
        <E>17394.268</E>
        <L>0.036</L>
        <T>00:16:06</T>
        <MaxND>3</MaxND>
        <MIwoc>100</MIwoc>
        <MIcwc>20</MIcwc>
        <MI>120</MI>
        <operators>
                <token count="1">#</token>
                <token count="3">()</token>
                <token count="1">+</token>
                <token count="1">++</token>
                <token count="1">-</token>
                <token count="2">--</token>
                <token count="2">:</token>
                <token count="2">::</token>
                <token count="12">;</token>
                <token count="3">=</token>
                <token count="1">==</token>
                <token count="7">&gt;</token>
                <token count="1">?</token>
                <token count="2">break</token>
                <token count="4">case ...:</token>
                <token count="1">default</token>
                <token count="1">else</token>
                <token count="3">if()</token>
                <token count="3">return</token>
                <token count="1">switch()</token>
                <token count="8">{}</token>
                <token count="4">||</token>
        </operators>
        <operands>
                <token count="5">0</token>
                <token count="1">1</token>
                <token count="1">10</token>
                <token count="1">100</token>
                <token count="1">2</token>
                <token count="1">200</token>
                <token count="1">3</token>
                <token count="1">5</token>
```

```
                        <token count="1">6</token>
                        <token count="1">7</token>
                        <token count="1">8</token>
                        <token count="1">9</token>
                        <token count="2">MyClass</token>
                        <token count="1">SomeFunction</token>
                        <token count="13">a</token>
                        <token count="1">foo1</token>
                        <token count="1">foo2</token>
                        <token count="10">i</token>
                        <token count="1">include</token>
                        <token count="6">int</token>
                  </operands>
            </file_total>
      </file>
      <system>
            <files>4</files>
            <functions>13</functions>
            <LOCphy>271</LOCphy>
            <LOCbl>45</LOCbl>
            <LOCpro>116</LOCpro>
            <LOCcom>113</LOCcom>
            <semicolons>51</semicolons>
            <vG_b>12</vG_b>
            <vG_e>16</vG_e>
            <params>4</params>
            <MIwoc>107</MIwoc>
            <MIcw>40</MIcw>
            <MI>147</MI>
            <alarms>
                  <file_vG>
                        <measured>4</measured>
                        <alarmed>0</alarmed>
                        <percent>0</percent>
                        <lowlimit>1</lowlimit>
                        <highlimit>100</highlimit>
                  </file_vG>
                  <file_LOCpro>
                        <measured>4</measured>
                        <alarmed>0</alarmed>
                        <percent>0</percent>
                        <lowlimit>4</lowlimit>
                        <highlimit>400</highlimit>
                  </file_LOCpro>
                  <file_comment_percent>
                        <measured>4</measured>
                        <alarmed>0</alarmed>
                        <percent>0</percent>
                        <lowlimit>30</lowlimit>
                        <highlimit>75</highlimit>
                  </file_comment_percent>
                  <file_V>
                        <measured>4</measured>
                        <alarmed>0</alarmed>
                        <percent>0</percent>
                        <lowlimit>100</lowlimit>
                        <highlimit>8000</highlimit>
                  </file_V>
                  <file_B>
                        <measured>4</measured>
                        <alarmed>0</alarmed>
                        <percent>0</percent>
                        <lowlimit>0</lowlimit>
                        <highlimit>2</highlimit>
                  </file_B>
```

```
                        <file_MI>
                              <measured>4</measured>
                              <alarmed>0</alarmed>
                              <percent>0</percent>
                              <lowlimit>65</lowlimit>
                        </file_MI>
                        <function_vG>
                              <measured>13</measured>
                              <alarmed>0</alarmed>
                              <percent>0</percent>
                              <lowlimit>1</lowlimit>
                              <highlimit>10</highlimit>
                        </function_vG>
                        <function_LOCpro>
                              <measured>13</measured>
                              <alarmed>2</alarmed>
                              <percent>15</percent>
                              <lowlimit>4</lowlimit>
                              <highlimit>40</highlimit>
                        </function_LOCpro>
                        <function_comment_percent>
                              <measured>13</measured>
                              <alarmed>2</alarmed>
                              <percent>15</percent>
                              <lowlimit>30</lowlimit>
                              <highlimit>75</highlimit>
                        </function_comment_percent>
                        <function_V>
                              <measured>13</measured>
                              <alarmed>2</alarmed>
                              <percent>15</percent>
                              <lowlimit>20</lowlimit>
                              <highlimit>1000</highlimit>
                        </function_V>
                        <function_MI>
                              <measured>13</measured>
                              <alarmed>0</alarmed>
                              <percent>0</percent>
                              <lowlimit>65</lowlimit>
                        </function_MI>
                  </alarms>
                  <error_messages count="0">
                  </error_messages>
            </system>
</cmt_long_report>
```

The long report can be rather long. It is just and just human readable. Primarily this report is meant to be used by some XML add-on utility, which reads the measurements and picks those measures that are of interest and further processes them as the user wishes.

If this report had been generated with option -l only (not -lf), the sections <operators>…</operators><operands>…</operands> would just be absent. And should there had been -s (summary) option, the <function name "*fname()*">…</function> would just be absent.

The Excel output could have been produced as follows:

```
cmt -x -o xreport.txt stack.h demofile.h stack.cpp demofile.cpp
```

and the *xreport.txtt* report file would be as follows:

```
File;Line;Measured_object;vG_b;vG_e;Params;MaxND;LOCphy;LOCbl;LOCpro
;LOCcom;V;B(x100);T;N1;N2;n1;n2;D;E;L(x1000);MIwoc;MIcw;MI;WarnMask
"demofile.h";8;"MyClass::MyClass()";1;1;0;1;3;0;3;0;20;0;00:00:02;4;
3;4;3;2;39;500;137;0;137;1010
"demofile.h";11;"MyClass::~MyClass()";1;1;0;1;3;0;2;0;20;0;00:00:02;
4;3;4;3;2;39;500;137;0;137;1000
"stack.cpp";32;"Stack::Stack()";1;1;0;1;18;1;6;11;56;1;00:00:12;9;8;
5;5;4;226;250;103;47;150;0
"stack.cpp";51;"Stack::~Stack()";1;1;0;1;12;1;4;7;27;1;00:00:06;6;3;
6;2;5;121;222;113;46;160;0
"stack.cpp";64;"Stack::clear()";1;1;0;1;12;1;4;7;33;1;00:00:04;5;5;5
;5;3;83;400;112;46;159;0
"stack.cpp";77;"Stack::height()";1;1;0;1;12;1;4;7;29;1;00:00:03;5;4;
5;4;3;71;400;113;46;159;0
"stack.cpp";90;"Stack::push()";3;3;1;2;24;1;14;9;288;10;00:04:28;31;
29;15;13;17;4826;60;89;41;130;0
"stack.cpp";115;"Stack::pop()";2;2;0;1;18;4;7;7;94;3;00:00:41;12;11;
10;7;8;739;127;100;41;141;0
"stack.cpp";134;"Stack::top()";2;2;0;1;17;3;7;7;100;3;00:00:41;12;12
;10;8;8;751;133;101;42;143;0
"stack.cpp";152;"Stack::element()";1;1;1;1;1;0;1;0;52;1;00:00:11;7;7
;7;6;4;212;245;150;0;150;10
"demofile.cpp";9;"MyClass::foo1()";2;6;1;2;6;0;6;0;164;4;00:01:26;19
;19;10;10;10;1560;105;114;0;114;0
"demofile.cpp";16;"MyClass::foo2()";2;2;1;2;8;0;8;0;113;3;00:00:58;1
5;12;11;7;9;1062;106;112;0;112;0
"demofile.cpp";25;"SomeFunction()";5;5;0;3;17;0;17;0;209;7;00:02:55;
28;17;16;9;15;3158;66;96;0;96;100
```

The first row specified the column headers and the next rows have the column values. Because the rows are long, they show here as 2 lines. Further, the report is generated so that the excel field separator is ';' while it is by default the \t (tab) character.

With command

```
cmt -x -s -o xsreport.txt stack.h demofile.h stack.cpp demofile.cpp
```

the following Excel data file *xsreport.txt* would result

```
File;Line;Measured_object;vG_b;vG_e;Params;MaxND;LOCphy;LOCbl;LOCpro
;LOCcom;V;B(x100);T;N1;N2;n1;n2;D;E;L(x1000);MIwoc;MIcw;MI;WarnMask
"stack.h";60;"stack.h";2;2;0;0;60;12;22;29;343;9;00:03:58;38;30;15;1
8;13;4288;80;74;44;118;0
"demofile.h";18;"demofile.h";1;1;0;1;18;3;12;3;156;4;00:01:04;20;19;
7;9;7;1153;135;113;30;142;100
"stack.cpp";152;"stack.cpp";5;5;2;2;152;25;49;78;972;41;00:40:35;91;
83;25;23;45;43836;22;98;45;143;0
"demofile.cpp";41;"demofile.cpp";7;11;2;3;41;5;33;3;620;22;00:16:06;
64;51;22;20;28;17394;36;100;20;120;100
```

Here you have only five rows, the column header row and the four file summary rows.

Of the McCabe cyclomatic number two values, vG_b (*basic*) and (extended) vG_e, are calculated and shown Further these values can be *modified* or nor depending what has been the configuration file cmt.ini MCCABE_PREFERENCE setting.

The B value is represented as multiplied with 100 and the L value with 1000, none of the numerical values have a decimal separator. When also the values are separated by a semicolon (vs, say, with a comma), you can directly take this file as input to your Excel regardless of what your Excel assumes as decimal separator (comma or period). The used field separator is determined by EXCEL_FIELD_SEPARATOR in cmt.ini file.

The last field on each row is “WarnMask”. It contains an encoding on what measures were warned at the measured object. The encoding is represented as an integral value as follows:

| | |
|---|---|
| 0 | No warnings |
| Add 1 | McCabe warned |
| Add 10 | LOCpro warned |
| Add 100 | Comment percent warned |
| Add 1000 | Halstead V warned |
| Add 10000 | Halstead B warned |
| Add 100000 | Maintainability Index warned |

For example, WarnMask value 10010 would mean that Halstead B and LOCpro are warned (are outside of the acceptable ranges).

### 5.7. Using cmt2html Utility

The *cmt2html* utility is used to convert a short tabular CMT++ report (the default report form) into a hierarchical color-coded html representation, which can be viewed with commonly used html browsers. It is a command line utility, effectively a Perl script, whichhas the following command line options:

**cmt2html -h**
**cmt2html** [**-i** *inputfile*] [**-o** *outputdir*] [**-s** *sourcedir*]…

[**-l** *splitsize*] [**-no-html-sources**] [**-no-java-script**] [**-nsb**] [**-p prefix**]

where

**-h**

Displays a command line help of the options.

**-i** *inputfile*

(optional) Specifies the input file that is read by the tool. The input file must be a CMT++ Complexity Measures Report, so called short tabular form, such that has been produced with *cmt* and when no –l, -lf, -x, -s options have been applied. If no –i option is given, *stdin* is read.

**-o** *outputdir*

(optional) Specifies the directory, which will be created if needed, and where the generated html files are written.When no –o is given, directory CMTHTML in current directory will be used.

**-s** *sourcedir*

(optional) Specifies one directory (but there can be many -s options) where *cmt2html* searches for the source files to be able to make a html'ized copy of them (unless denied with **-no-html-sources** option) or just to construct a html link to them (when **-no-html-sources** option is given).

The rule, in order, for finding the source file is the following (Assume input CMT++ report has "..\dir2\file.c" and in -s option it is given "d:\tempdir"):

(1) If the source file is found with the name that it has on input CMT++ report (..\dir2\file.c), these **–s** options have no effect on the treatment of that source file.
(2) If not yet found, file d:\tempdir\..\dir2\file.c is looked.

Repeated on each –s option argument
(3) If not yet found, file d:\tempdir\file.c is looked. Repeaetd on each **–s** option argument.
(4) If still not yet found, in the generated html report there is link on this file to a page saying that the source file was not found.

**-l** *splitsize*

(optional) Specifies how big html junks are generated for the detailed report html page. Whenever at the beginning of a new source file the detailed html page contains already over *splitsize* lines, a new html page is started and linked with *next* and *previous* links to its neighbors. Splitting the detailed html report may be needed (for the sake of faster load times), if the html report contains much data (hundreds of source files, thousands of functions, ...). The default *splitsize* is 1000 lines.

**-no-html-sources**

(optional) Determines that no html'ized copies of the source files are generated in the output directory. Only links to them are generated with such path and file names as they could be resolved at the time and context of running cmt2html.

**-no-javascript**

(optional) Do not generate Javascript calls to the resultant HTML pages. Some browsers may have them disabled. The function source code display gets also disabled.

**-nsb**

(optional) “No Start Browser”, advises *cmt2html* that browser is not started on the generated html representation, only the html files are generated. This option has effect only on Windows environment, where only the default/automatic starting of the browser is supported.

**-p** *prefix*

(optional) Determines how the generated html files are named in the output directory. They are named as *prefix*.html, *prefix*A.html, *prefix*B.html, etc. In the absence of this option the default prefix is *index.* (Now, as of CMT++ v5.0, when there is the **–o** option, you might consider this **–p** option as deprecated. However it is kept here for compatibility reasons.)

The *cmt2html* utility does the following:

- Reads the CMT++ Execution Profile listing that is given with the **–i** option. The input file must be of “short tabular form.
- Creates, if needed, the output directory (default is CMTHTML in current directory) and writes there (blindly overwrites any possible previous files with same name) a number of html files making up the html representation of the input file. Unless denied with **-no-html-sources** option, also the html'ized copies of the C/C++ source files are constructed.
- The file where the browsing is started is *prefix*.html (by default index.html) at that directory.

When the html representation is opened with a browser a *summary window* is shown first. It contains the file-level summaries with a histogram of the percent of alarms at the files. The data lines represent the file summary level, as if *cmt* had been run with **–s** option. Red color indicates that there have been alarms at the file.

The column header titles “Alarms-%”, “v(G)”, “LOCphy”, “LOCpro”, “V”, “B” and “MI”(or “MIwoc”) are links to a sorted view on that specific measure.

The file name serves as a link to the *detailed window*. Clicking on it opens a new window and positions it to the detailed CMT++ listing of the file. The *detailed window* is effectively as the input file, i.e. the CMT++ Complexity Measures Report, but made to a

color-coded html form. If **–l** *splitsize* option was used, the *detailed window* may be in a number of smaller html junks, which load faster and which are linked together with *next* and *previous* links.

In the detailed window the source file name is a link. By default it points to the html'ized source file copy in the output directory, and shows the whole source file. If **-no-html-sources** option was used, it is a link to the actual source file, and it is up to the browser that you use how a C/C++ source file gets shown. If the source file was not found, the source file name is a link to a page, which tells that source file was not found at cmt2html time.

In the detailed window the function names are by default (when option **-no-javascript** was not given) links, too. Clicking on the function name brings to the screen the lines from the source file that make up the corresponding function. The lines vanish from the screen by clicking the name again.

This showing the source code of the individual functions is implemented by Javascripts in the generated html. If the html has been generated in the default way, but your browser does not allow Javascripts, you can experience various warning messages from the browser, and these links do not function at all or they do not work as intended (depending on the browser). When **-no-javascript** option was used, the function names are not links at all, the generated html does not contain Javascipts, and you do not get any complaints from your browser, if it has Javascripts disabled.

The html representation gives you a fast way to browse the CMT++ Complexity Measures Report at summary level, at detailed level and all the way at the whole source file/individual function level.

The *cmt2html* utility is effectively a Perl script. On Unixes it is assumed the Perl is available there already. On Windows platform the utility uses a Perl interpreter that comes along with the CMT++ delivery package (will reside at %CMTHOME%\perl).

An example:

```
C:\SOURCEDIR> cmt –o report.txt somefiles*.h somefiles*.cpp
C:\SOURCEDIR> cmt2html –i report.txt -nsb
C:\SOURCEDIR> start CMTHTML\index.html
```

# 6. Interpreting Complexity Measures

This chapter discusses how the measurement results of CMT++ can be applied. It is not possible to give absolute limits to acceptable values. The limits given and explained below are common suggestions. These suggestions are based on measurements made on code maintained with good success. You can configure CMT++ for project specific needs by changing the limit definitions in the configuration file.

## 6.1. Lines-of-Code Metrics

Lines-of-code metrics are the most traditional measures used to quantify software complexity. They are simple, easy to count, and very easy to understand. They do not, however, take into account the intelligence content and the layout of the code. CMT++ calculates the following lines-of-code metrics:

- LOCphy: number of physical lines
- LOCbl: number of blank lines (a blank line inside a comment block is considered to be a comment line)
- LOCpro: number of program lines (declarations, definitions, directives, and code)
- LOCcom: number of comment lines

A line containing both program code and a comment is counted to both LOCpro and to LOCcom.

The following recommendations are given for the lines-of-code measures. See the configuration file (*cmt.ini*) on what of these recommendations can be fine-tuned. CMT++ will mark an alarm in the case that the measured value is outside the recommended bounds.

Function length should be 4 to 40 program lines. A function definition contains at least a prototype, one line of code, and a pair of braces, which makes 4 lines. A function longer than 40 program lines probably implements many functions. Functions containing one selection statement with many branches are an exception to this rule. Decomposing them into smaller functions often decreases readability.

File length should be 4 to 400 program lines. The smallest entity that may reasonably occupy a whole source file is a function, and the minimum length of a function is 4 lines. Files longer than 400 program lines (10..40 functions) are usually too long to be understood as a whole.

At least 30 percent and at most 75 percent of a file should be comments. If less than one third of a file is comments the file is either very trivial or poorly explained. If more than 75% of a file are comments, the file is not a program but a document. In a well-documented header file percentage of comments may sometimes exceed 75%.

### 6.2. Cyclomatic number.

The cyclomatic number v(G) describes the complexity of the control flow of the program. For a single function, v(G) is one less than the number of conditional branching points in the function. The greater the cyclomatic number is the more execution paths there are through the function, and the harder it is to understand. Note, that the cyclomatic number is insensitive to the complexity of data structures, data flows, and module interfaces.

CMT++ supports a couple of flavors of cyclomatic number. The selection is done by configuration file cmt.ini setting MCCABE_PREFERENCE. . The options are:

*basic*:

Operators &&, || are not calculated to v(G).

*extended*:

Operators &&, || are calculated to v(G).

*basic_modified*:

Like *basic*, but ‘case n:’ labels of a ‘switch(…){…}’ statement are not calculated to v(G), but the ‘switch(…)’ itself is calculated as +1 to v(G).

*extended_modified*:

Like *extended*, but ‘case n;’ labes of a switch(…){…} statement are calculated to v(G), but the ‘switch(…)’ itself is calculated as +1 to v(G).

In our industry there are advocates on each of these ways to calculate v(G). The defaut setting in configuration file is *extended*. It also represents the most puritanistic view to the matter: number of conditional branches in the code, and normally gives highest v(G) values.

The cyclomatic number of a function should be at most 10. If a function has a cyclomatic number of 10, there are at least 10 (but probably more) execution paths through it. More than 10 paths are hard to identify and test. Functions containing one selection statement with many branches make up an exception.

A reasonable upper limit Cyclomatic number of a file is 100.

The function (including class/struct declaration) and file level Cyclomatic number recommended low and high levels are defined in configuration file (*cmt.ini*).

If you are measuring non-preprocessed source files, as presumably is the normal case, CMT++ considers the conditional compilation directives (like #ifdef, and, if *extended/extended_modified*, the && and || operators in them) to increase the v(G) as well.

Technically CMT++ calculates both *basic* (or *basic_modified*) and *extended* (or *extended_modified*) cyclomatic numbers. They are reported in Excel and XML form reports with names vG_b and vG_e.

### 6.3. Maximun nesting depth

Somewhat related to v(G) measure is the MaxND measure. It is the maximum nesting depth of {} braces in a function body. 5 is a reasonable upper limit for MaxND. This is reported in Excel and XML form reports.

### 6.4. Number of function parameters

Number of function parameters is calculated and reported in Excel and XML form reports.

### 6.5. Volume (V)

Halstead's volume V describes the size of the implementation of an algorithm. The computation of V is based on the number of operators and operands (distinct and total) in the algorithm. Therefore V is less sensitive to code layout than the lines-of-code measures.

The volume of a function should be at least 20 and at most 1000. The volume of a parameterless one-line function that is not empty, is about 20. A volume greater than 1000 tells that the function probably does too many things.

The volume of a file should be at least 100 and at most 8000. These limits are based on volumes measured for files whose LOCpro and v(G) are near their recommended limits. The limits of volume can be used for double-checking.

### 6.6. Estimate for Delivered Bugs (B)

Halstead's delivered bugs B is an estimate for the number of errors in the implementation.

Delivered bugs in a file should be less than 2. Experiences have shown that, when programming with C or C++, a source file almost always contains more errors than B suggests. The number of defects tends to grow more rapidly than B.

In tool version v4.1 it was introduced configuration setting B_CORRECTION_FACTOR. . It is a positive decimal number and has default value 1.0. The initial B value (as calculated by the original Halstead formulae) is multiplied with B_CORRECTION_FACTOR before reporting and comparing to alarm limits. With this setting you can fight against the hypothesis that the original Halstead B measure estimates the number of bugs to too low. For example, B_CORRECTION_FACTOR=1.95 might be a reasonable value, if you decide to take this policy.

### 6.7. Maintainability Index (MI/MIwoc)

Maintainability Index (MI) was introduced to software engineering in 1992 at the International Conference on Software Maintenance, in a presentation given by Paul Oman and Jack Hagemeister. Currently many measurement tools provide this measure.

Maintainability Index is calculated with certain formulae (see "Appendix B. How the Measures Are Calculated") from lines-of-code measures, McCabe extended cyclomatic number and Halstead measures. MI is a composite measure, which strives to express the relative maintainability of a complete software system in a single number, which is straightforward to calculate and which would have good predictive value.

There are two variants of Maintainability Index: one that is with comments (MI) and one that is without comments (MIwoc). CMT++ calculates them both. Which one is shown in some of the basic CMT++ text form report is selected by configuration file setting MI_PREFERENCE. .

Much of the whole idea of the MI is to derive a single number, the Maintainability Index, of the whole software system. CMT++ calculates such a number over all the code that is inputted to it in one cmt run. In CMT++ that number is called system-level MI. CMT++ also calculates file-level MI and function-level MI.

In the general MI modeling the 'system' is considered to be a collection of 'modules'. In CMT++ the 'module' is taken to be a function definition. So, only those executable code snippets are noticed when calcultating the system-level MI number, which is

the main interest in the MI modeling. This means that the code that is in header files and the code that is in files between the functions is ignored. This is explained so that the "ignored code" anyway gets referred to in the function bodies, which are the actual execution entities of the system. The "ignored code" gets noticed indirectly in that way.

See also chapter "Appendix B. How the Measures Are Calculated" for more discussion about the MI.

Maintainability Index (MI, with comments) value 85 and over suggests good maintainability. Values 65 – 85 suggest moderate maintainability. Values below 65 suggest that the system is difficult to maintain. With really bad pieces of code (big, uncommented, unstructured) the MI value can be even negative.

Whether you follow MI or MIwoc depends on how much you trust on the validity of the commenting in the code. If the comments are very out-of-date or if they are just some standard header blocks with no real value to the maintainer, you might want to follow the MIwoc measure instead of MI. And, of course, with this MI measure and with all the other CMT++ measures, you should use your common sense and make some experimental measurements with some familiar software for finding the best-suited alarm limits.

## 6.8. Complexity, Quality Assurance, and Testing

The more complex a module is, the more likely it is to contain errors, and the more difficult it is to test. The more complex it is, the harder it is to maintain. The more it contains errors, the more it needs to be maintained. The problem feeds itself! Some algorithms are complex, regardless of the way they are implemented, but in most cases complex implementation is due to bad design.

Source code quality assurance usually utilizes teamwork methods like code reading, structured walkthroughs, and inspections. These methods are invaluable; a tool can never find all those kinds of errors that a human inspection can. In addition to that, the team inspections also help the members of the team to learn from each other. However, there is usually too little time to inspect all the

code carefully. In such a case, it is important to select the most important and most error-prone modules to the inspection.

Obviously, the modules that have high complexities need to be inspected most carefully. How high "high" is depends, of course, on your development process and quality criteria. The default alarm limits of CMT++ are a good starting point. If you have collected information about error densities of your code, you can measure complexities of those codes and draw your own conclusions about suitable alarm limits.

When dynamic testing is concerned, the most important complexity measures are the cyclomatic number (v(G)) and the number of delivered bugs (B). Because the cyclomatic number describes the control flow complexity, it is obvious that modules and functions having high cyclomatic number need more test cases than modules having a lower cyclomatic number. As a rule of thumb, each function should have at least as many test cases as indicated by its cyclomatic number. The number of delivered bugs approximates the number of errors in a module. As a goal at least that many errors should be found from the module in its testing.

When assessing big amounts of foreign code, the Maintainability Index is perhaps the most simple and fast to use indicator of the code “level” in maintainability perspective.

# Appendix A. The Source Code Language

This appendix discusses how CMT++ analyzes the source language.

CMT++ operates on C, C++ and C#. CMT++ consideres C to be a subset of C++. Using C++ keywords as symbols in C programs may cause a small error to the calculations. CMT++ also recognizes the extensions, such as the keywords *_near*, *_far* and *_huge*, of some commonly used C/C++ compilers in PC environment (Microsoft, Borland). It is often also possible to measure other beyond-ANSI dialect C/C++ programs with CMT++, but then CMT++ may not be able to recognize all measured items correctly.

C# code largely resembles C++ code, but it has some language constructs, e.g. *get{…}* and *set{…}* property declarations, that C++ does not have. When CMT++ knows that the code is C# (ref. "`4.3. C# Code Handling Parameters`") the C#-specific language constructs are identified and reported properly.

CMT++ does not actually check the input file syntax for being correct C/C++/C#. CMT++ assumes that the input file compiles correctly with your compiler. However, it is possible to measure also erroneous or incomplete code, provided that the block structure is correct.

When C/C++ code it is assumed to be fed to CMT++ in non-C-preprocessd form. Thus CMT++ derives the measures of the program file level that the programmer sees, edits and maintains.

CMT++ accepts the C-preprocessor directives, but it does not process the source code according to the directives. It is also possible to C-preprocess the source files before passing them to CMT++. This will normally result in higher complexity values, because macros are expanded and include directives are replaced with the contents of the include files. In some cases analyzing

preprocessed code can result in lower complexity, because conditional compilation has removed some code sections.

The C and C++ C-preprocessor operates on character level. It does not recognize the block structure of the source language. Therefore it is possible to write macros, which expand to only a part of a block or a token. Programs using the C-preprocessor in such an 'unstructured' way must in general be C-preprocessed before passing to CMT++.

CMT++ recognizes and processes /* ... */ block comments and // ... line comments.

If you still happen to have old Kernighan-Ritchie 1 type of C code, i.e. something like

```
int foo()
   int par1;
   int par2;
{
   return par1 + par2;
}
```

Note that these kinds of functions are not recognized nor reported separately by CMT++. Here the foo() measures are counted only to file level figures.

# Appendix B. How the Measures Are Calculated

The complexity measures are calculated for the following parts of the source program:

- For each function definition (function body)
  - Stand-alone function
  - Member (inline) function defined inside a class or struct declaration
  - Member function defined separately outside the class or struct declaration
- For each source file
- For all source files together ("system level") lines of code measures, number of semicolons, McCabe's v(G) and Maintainability Index measures.

## B.1. Lines of code Metrics

The lines of code measures are calculated according to the following rules:

- Blank lines (LOCbl) are the lines containing only white-space characters.
- Comment lines (LOCcom) are the lines containing C or C++ comments. A blank line inside a /* ... */ block comment is counted to comment lines.
- Program lines (LOCpro) are the lines that contain program code. Preprocessor directives and macro definitions are considered also to be program lines. A line is counted to LOCpro and to LOCcom, if it has both program code and comment. Lines containing preprocessing directives are calculated to LOCpro.
- Physical lines (LOCphy ) are the lines that there are in a text file sense in the measured fragment of code.

When calculating the lines of code metrics for a function, the configuration parameter NOTICE_LEADING_COMMENTS determines whether the immediately preceding consecutive comment block is associated to function lines or not. For example, assume we have the following code fragment:

```
...
/* Local functions: */

/* Function foo does something
   simple
*/

int foo(int arg) {
   return arg++; /* returning one bigger */
}
...
```

When NOTICE_LEADING_COMMENTS=1, for foo() it is calculated: LOCphy=7, LOCpro=6, LOCcom=4, LOCbl=1. But with NOTICE_LEADING_COMMENTS=0, for foo() it is calculated: LOCphy=3, LOCpro=3, LOCcom=1, LOCbl=0, and the preceding comment block is calculated only to the file-level counts.

### B.2. Halstead Metrics

Halstead's metrics is based on interpreting the source code as a sequence of tokens and classifying each token to be an operator or an operand. Then it is counted

- **number of unique operators** (n1)
- **number of unique operands** (n2)
- **total number of operators** (N1)
- and **total number of operands** (N2)

in the piece of source code under measurement (function or file). Other Halstead measures are derived from these with certain fixed formulas as described later.

Halstead measures are independent on comments and what is the code “layout” in the source file.

No general, language independent rule exists for classifying tokens to operators and operands. Before describing how CMT++ makes the operator/operand distinction, an auxiliary classification of tokens used by CMT++ is described:

IDENTIFIER All identifiers that are not reserved words.

SCSPEC (storage class specifiers) Reserved words that specify storage class: *auto, extern, inline, register, static, typedef, virtual, mutable.*

TYPESPEC (type specifiers) Reserved words that specify type: *bool, char, double, float, int, long, short, signed, unsigned, void, wchar_t***.** This class also includes some compiler specific nonstandard keywords.

TYPE_QUAL (type qualifiers) Reserved words that qualify type: *const, constexpr, friend, volatile*.

RESERVED Other reserved words: *asm, break, case, class, continue, default, delete, do, else, enum, for, foreach, goto, if, new, operator, private, protected, public, return, sizeof, struct, switch, this, union, while, namespace, using, try, catch, throw, const_cast, static_cast, dynamic_cast, reinterpret_cast, typeid, template, explicit, true, false, typename, abstract, as, base, checked, decimal, event, finally, fixed, internal, is, lock, null, object, out, override, params, readonly, ref, sbyte, realed, stackalloc, unchecked, unsafe*. This class also includes some compiler specific nonstandard keywords.

Same set of identifiers is recognized to be reserved words regardless if the code to be measured is C, C++ or C#. But in practical code e.g. usage of a C++ keyword as variable name in C code is quite rare, and so the bias to operator/operand classification (discussed below) is considered to be marginal.

| | |
|---|---|
| CONSTANT | Character, numeric or string constants. |
| OPERATOR | One of the following: ! != % %= & && \|\| &= ( ) * *= + ++ += , - -- -= -> . ... / /= : :: < << <<= <= = == > >= >> >>= ? ?? [ ] ^ ^= { } \| \|= ~ ; # |
| COMMENTS | The comments delimited by **/*** and ***/** or // and newline do not belong to the set of C++ tokens but they are counted by CMT++. |

The classification rules of CMT++ are determined so that frequent language constructs give intuitively sensible operator and operand counts. For example the statement f(x, y) is counted as follows: f, x and y are operands and the parentheses and comma are operators. You can see how CMT++ does the source code token classification into operands and operators by taking the report with **-lf** option

Tokens of the following categories are all counted as operands by CMT++: IDENTIFIER, TYPENAME, TYPESPEC, CONSTANT.

Tokens of the following categories are all counted as operators by CMT++: SCSPEC, TYPE_QUAL, OPERATOR, RESERVED, preprocessor directives. However, the tokens *asm* and *this* are counted as operands.

The following control structures are treated in a special way:

| | |
|---|---|
| *case ...:* | The colon is considered to be a part of the case construct. The case and the colon are counted together as one operator. |
| *for (...)* | (Also *foreach(…)*, possible in C# code) The parentheses are considered to be a part of the for construct. The for and the parentheses are counted together as one operator. |
| *if (...)* | The parentheses are considered to be a part of the if construct. The if and the parentheses are counted together as one operator. |

*switch (...)* The parentheses are considered to be a part of the switch construct. The switch and the parentheses are counted together as one operator.

*while (...)* The parentheses are considered to be a part of the while construct. The while and the parentheses are counted together as one operator.

*catch (...)* The parentheses are considered to be a part of the catch construct. The catch and the parentheses are counted together as one operator.

The number of unique operators and operands (n1 and n2) as well as the total number of operators and operands (N1 and N2) are calculated by collecting the frequencies of each operator and operand token of the source program. All other Halstead's measures are derived from these four quantities using the following set of formulas.

The **program length** (N) is the sum of the total number of operators (N1) and operands (N2) in the program:

$$N = N1 + N2$$

The **vocabulary size** (n) is the sum of the number of unique operators (n1) and operands (n2):

$$n = n1 + n2$$

The **program volume** (V) is the information contents of the program, measured in mathematical bits. It is calculated as the program length times the 2-base logarithm of the vocabulary size:

$$V = N * \log2(n)$$

The **difficulty level** or error proneness (D) of the program is proportional to the number of unique operators in the program. D is also proportional to the ratio between the total number of operands and the number of unique operands (i.e. if the same operands are used many times in the program, it is more prone to errors).

$$D = (n1/2)*(N2/n2)$$

The **program level** (L) is the inverse of the error proneness of the program. I.e. a low level program is more prone to errors than a high level program.

L = 1/D

The **effort to implement** (E) or understand a program is proportional to the volume and to the difficulty level of the program.

E = V * D

The **time to implement** or understand a program (T) is proportional to the effort. Empirical experiments can be used for calibrating this quantity. Halstead has found that dividing the effort by 18 gives an approximation for the time in seconds.

T = E/18

The **number of delivered bugs** (B) correlates with the overall complexity of the software. Halstead gives the following formula for B (below "**" stands for "to the exponent"):

B = (E**(2/3))/3000

Before reporting the B value is multiplied with B_CORRECTION_FACTOR. )

Both at function and at file level the Halstead measures are calculated in the above described way.

Halsetad measures are calculated and reported on functions and files, no more at all-files-together level.

### B.3. McCabe Metrics

McCabe Cyclomatic number v(G) is calculated on (standalone and member) function definitions (“bodies”), on source files and on all-files-together level.

McCabe's Cyclomatic number v(G) shows the complexity of the flow of control through a piece of code. v(G) is the number of conditional branches in the flowchart. v(G) = 1 for a program

consisting of only sequential statements, no conditional branching in it.

Each if-statement introduces a new branch to the program and therefore increases v(G) by one. Iteration constructs such as for- and while-loops also introduce branches. Each case ...: part in the switch-statement increase the v(G) by one. The default: case branch does not increase v(G), because it does not increase the number of branches in the control flow. If there are two or more case ...: parts that have no code in between, the McCabe measure is increased only with one for all those case ...: parts. Each catch (...) part in a try-block increases v(G) by one. Construction expr1 ? expr2 : expr3 increases v(G) by one.

It should be noted that v(G) is insensitive to unconditional branches like goto-, return- and break-statements although they surely increase complexity.

In CMT++ the branches generated by conditional compilation directives are also counted to v(G). Even if conditional compilation directives do not add branches to the control flow of the executable program, they increase the complexity of the program file that the user sees and edits.

In CMT++ v5.0 there came possibility to calculate McCabe cyclomatic number in one of the following “flavors”: *basic, extended, basic_modified, extended_modified*. What calculation rule is used is determined by MCCABE_PREFERENCE configuration setting. Default flavor is *extended*, which is the one how CMT++ calculated v(G) also before v5.0.

In summary, the following language constructs increase (or can increase) the cyclomatic number by one: *if (...), for (...), foreach(…), while (...), switch(…), case ...:, catch (...), &&, ||, ?, ??, #if, #ifdef, #ifndef, #elif.*

Depending in what “flavor” the v(G) is calculated, there are some differences. As an example consider the following function:

```
int foo(int a, int b, int c) {
   if (a == 5 && b == 6) {
      some_statements1;
      return 7;
   }
```

```
   switch (c) {
   case 1:
      some_statements2;
      break;
   case 3:
   case 4:
   case 6:
      return c + 1;
   case 8:
      some_statements3;
   default:
      some_statements4;
   }
   return 8;
}
```

When MCCABE_PREFERENCE=*basic*: && and || do not give additional v(G) points. In the above function v(G) is 5. They come as follows: 1 (start initially) + 1 (of if) + 1 (of case 1:) + 1 (of the consecutive cases case 3:, case 4:, case 6:) + 1 (of case 8:).

When MCCABE_PREFERENCE=*extended*: The default or traditional usage, each && and || give +1 to v(G). In the above function v(G) is 6. It is calculated in the same way as in *basic* case, but the && operator gives one +1.

When MCCABE_PREFERENCE=*basic_modified*: Like *basic*, but case n: labels do not increase v(G), instead switch() increases it. In the above function v(G) is 3.They come as follows: 1 (start initially) + 1 (of if) + 1 (of switch(…)).

When MCCABE_PREFERENCE=*extended_modified*: Like *extended* but case n: labels do not increase v(G), instead switch() increases it. In the above function v(G) is 4. It is calculated in the same way as in *basic_modified*, but the && operator gives one +1.

On functions the McCabe Cyclomatic number is calculated as described above.

On file level the McCabe Cyclomatic number is calculated according to following rules:

- Start calculations from 1.
- If the file contains function definitions, whose McCabe complexity is over one, add of each of them the amount how much they are over one. For example, if some function has

complexity 5, the file level complexity measure is increased by 4. If all the functions in a file have v(G) 1, also the whole file has v(G) 1 (unless some code between the functions gives some v(G) points to file level).

- If there is some code between function definitions or class/struct definitions (there might be conditional compilation directives like #ifdef), add that complexity to the file level measure.

On all files together level the McCabe Cyclomatic number is calculated according to the following rules:

- Start calculation from 1

If there are participating files, whose McCabe complexity is over one, add of each of them the amount how much they are over one. For example, if some file has complexity 20, the all files together complexity measure is increased with 19.

### B.4. Maximum nesting depth

Maximum nesting depth MaxND is calculated on functions. It is reported also on file level, where it means the maximum MaxND value of the file's functions.

MaxND is a measure on how deep is the maximum {} nesting in the function. For example, in the following code snippet

```
void foo1() {
.../* some code having no {}s */
}
void foo() {
   if(cond1) {
      if(cond2) {
         /* some code having no {}s */
      }
   }
   if(cond3) {
      /* some other code having no {}s */
   }
}
```

Here the MaxND is 1 for foo1 and 3 for foo2. In the counting only the explicit {}s are noticed that are in the function body .

The MaxND is somewhat similar to v(G), but gives another and supplementary view to the algorithmic complexity of the code.

### B.5. Maintainability Index

Maintainability Index is calculated on each function, on each file and on all files together level. Actually there are three measures:

- MIwoc: Maintainability Index without comments
- MIcw: Maintainability Index comment weight
- MI: Maintainability Index = MIwoc + MIcw.

The general formulae for MI is the following:

> MIwoc = 171 – 5.2 * ln(aveV) – 0.23 * aveG – 16.2 * ln(aveLOC)
>
> MIcw = 50 * sin(sqrt(2.4 * perCM))
>
> MI = MIwoc + MIcw.

Where

> aveV = average Halstead Volume (CMT++'s V) per module
>
> aveG = average *extended* cyclomatic complexity (CMT++'s v(G)) per module. (The v(G) value may be *modified* or not depending what there has been in MCCABE_PREFERENCE setting in the configuration file)
>
> aveLOC = average count of lines (CMT++'s LOCphy) per module
>
> "module" is (in CMT++ case) a C-like function definition or a C++-like member function definition.

As an example, consider the following simple 15-line file1.cpp:

```
// Some file level comments
#include "file1.h"
int gv = 0; //some file level code
```

```
// Function header comment
int foo() {
   if (gv < 0) gv++; // line comment
   return gv;
}

void Aclass::bar() {
   gv = something;
}
```

At function level there is no “average” calculation. The divider is 1 straight away. For the above two functions (in MI modeling, modules) CMT++ calculates the following measures:

foo(): V=48, v(G)=2, LOCphy=5, LOCcom=2, MIwoc=124, MIcw=42, MI=166.

Aclass::bar(): V=33, v(G)=1, LOCphy=3, LOCcom=0, MIwoc=135, MIcw=0, MI=135.

At file level, however, CMT++ makes the calculation a bit un-orthogonally. The file1.cpp contains 2 modules (in the way as modules are understood in MI modeling), functions foo() and Aclass::bar(). However, at single file-level CMT++ does not calculate the MI measures via the averages of its participating modules but directly from file-level V, v(G), and LOC measures after having divided these file-level measures first with the number of modules. If there are no modules in the file, the divider is 1.

In this way the file-level code, which lexically is not included in any module, has its impact at the file-level MI measures. . Here, for the file1.cpp the file-level measures are: V=132, v(G)=2, LOCphy=15, LOCcom=4, MIwoc=116, MIcw=36, MI=152.

The all files together level is considered to be the whole Software or just system-level. For it CMT++ calculates the MI measures via the participating module averages. In that calculation the code that is lexically outside of modules has no effect to the system-level MI measure. Such code has indirectly some effect (via module’s Halstead V measure), because the variables, types, macros, etc. are used in the modules.

Assuming that only this one file file1.cpp would be the “all files”, the MI measures are calculated from the following values:

V=(48+33)/2, v(G)=(2+1)/2, LOCphy=(5+3)/2, LOCcom=(2+0)/2, MIwoc=129, MIcw=35, MI=164.

It is the value 129 (contribution of comments excluded) or 164 (contribution of comments included), which is the single number estimating the maintainability of the whole software system.

# Appendix C. Measuring Assembly Code

As of the CMT++ v3.3 the tool has supported also assembly code measuring. Complete separate assembly files can be measured and assembly code, which is inside of a C/C++ file.

Perhaps only the lines-of-code measures (LOCphy, LOCpro, LOCcom and LOCbl) are meaningful with assembly code. CMT++ makes also some attempt to measure Halstead measures of assembly code, but the algorithm for it is a bit heuristic. CMT++ does not measure McCabe of assembly code.

The following configuration file parameters are used when measuring assembly code:

- ASSEMBLY_FILE_EXTENSIONS
- ASSEMBLY_ID_ADDON_CHARACTERS
- ASSEMBLY_COMMENT_CHARACTER

## C.1 Measuring Complete Assembly Files

CMT++ recognizes that the source file is an assembly file from its extension. For example, if we have in configuration file (or we give the setting with **-C** option from command line) ASSEMBLY_FILE_EXTENSIONS=asm,s,as and we give command

```
cmt file1.s file2.asm file3.cc
```

the two first files are considered to be assembly files and the third one a C/C++ file. When a complete assembly file is measured, the measures are collected per whole file only. Vs. in a C/C++ file the measures can be collected per functions and summarized per file.

## C.2 Measuring Assembly Code Inside A C/C++ File

Different C/C++ compilers support widely different ways in expressing inline assembly code in a C/C++ source file. The starting keyword is one of the following (CMT++ recognizes these): asm, _asm, __asm, __asm__ .

The typical use cases are the following:

- asm *one-liner-assembly-code*
- asm {
  *many-assembly-code-lines*
  }
- asm("*one-string-literal*"); // official C++ way
- asm (*C/C++-token-stream*); // GNU C

The two first cases are measured as assembly code, the two latter cases are measures normally, as if they were C/C++ code.

Some compilers support writing assembly blocks starting from #pragma asm (or plain #asm) line and ending on #pragma endasm (or plain #endasm) line. These are recognized as well.

## C.3 Recognizing A Comment from An Assembly Code

First it should be noted that if assembly code has /*...*/ block comments or // line comments, they are identified as in C/C++ code and calculated to LOCcom lines. Native assembly code has also its own commenting style. It is assumed to be a line comment, which starts with some indicated special character. CMT++ looks this special character from configuration parameter ASSEMBLY_COMMENT_CHARACTER. A typical value for it is ';'. When parsing assembly code, and when this specified character is met, the line end is considered to be comment.

For example, consider the following code fragment (inside a C/C++ file):

```
...
__asm nop
__asm nop ; comment here
```

```
__asm nop // comment here
__asm nop /* comment here */

__asm /* comment here

         which continues upto here */
__asm {
nop ; comment here
; assembly comment here
nop /* comment here */
}
```

. . . Should the assembly comment character be ';', it's occurrences are not calculated to the ';' count of C/C++ code.

## C.4 Parsing Assembly Identifiers

In assembly code the identifiers may have a wider set of allowed character than in identifiers in C/C++ code. CMT++ allows in C/C++ identifiers the following characters: A-Z, a-z, 0-9, _, $. When parsing assembly code, CMT++ consults the configuration parameter ASSEMBLY_ID_ADDON_CHARACTERS. If the character is one of the specified additional characters, it is associated to the assembly identifier (vs. the identifier would be split into two or more tokens). For example, if we have setting ASSEMBLY_ID_ADDON_CHARACTERS=.@ , and we have the following assembly code

```
...
.title "SomeTitle"
.include somefile.inc
...
.ref __SOME_NAME
...
MOV .S1, .S2
...
```

The tokens are identified as you would intuitively assume. Notably the '.' can be a part of an identifier. On the other hand, this "correct" identifier association is relevant primarily only for Halstead measure calculation.

## C.5 Lines-Of-Code Measuring from Assembly Code

The LOCphy, LOCpro, LOCcom, LOCbl measures are calculated normally. /*...*/ and // comments in assembly code are calculated as comments, too.

## C.6 McCabe Measuring from Assembly Code

CMT++ does not measure McCabe from assembly code.

## C.7 Halstead Measuring from Assembly Code

Halstead measure is based on the number of operators (N1), number of operands (N2), number of unique operators (n1) and number of unique operands (n2) in a piece of code. All other Halstead measures are arithmetic derivatives from these.

As a first rule, CMT++ categorizes the token stream from an assembly code into operands and operators in the same way as it does it with C/C++ code. The following heuristic rule is an exception, when assembly code is being processed: If the token is the first token on line or the token is preceded with asm (or _asm, __asm), the token is considered to be an operator.

Consider the following assembly code fragment:

```
...
__asm mov a, b
...
__asm {
   nop
   mov a, b
}
...
```

Here __asm, mov, nop, {} and ',' are operators. a and b are operands.

# Appendix D. cmt Error Messages

Each cmt error message takes one of the following forms:

```
*** CMT++ error error-code : While processing file
      file-name  around line line-number
   error-text
```

or

```
*** CMT++ fatal error error-code :
   error-text
```

or

```
*** CMT++ warning error-code : While processing file
      file-name  around line line-number
   error-text
```

where *error-code* is an integer value used to identify the error (helps in communicating with Testwell) and *error-text* is the actual error message. The message portion `While processing file` *file-name* `around line` *line-number* comes only if the message is related to processing of some file.

As a program cmt returns an exit code to the operating system level. The exit codes are the following:

0 cmt run ended normally with no error messages.

1 cmt run ended normally but some error messages were written.

2 cmt run ended to a fatal error and the run was aborted.

These messages are written to *stderr*.

The possible *error-texts* are the following:

`Bad combination of options on command line, type -h to get help`

For example there was -x and -l options at the same time.

`Bad –C option: *the_bad_option_value*`

Bad –C option value given on command line.

`Bad conf parameter '*xxx*'`

Configuration file contained a bad parameter definition.

`Bad definition *xxx* in configuration file`

Bad definition in configuration file

`Huge string literal, cutted`

The string is longer than CMT++ is prepared for; only the string begin is reported. (Warning only)

`Cannot create file *filename*`

Creation of the given file for CMT++ run output failed.

`Cannot open file *filename*`

The given source file could not be opened.

`CMT++ run aborted`

After searching all the locations for configuration parameters and noticing the possible **-c** option there still was one or more required configuration parameters unset. The previous message described the more detailed reason.

`Could not find configuration file *filename*`

The configuration file given explicitly in the **-c** option could not be opened, or, in the absence of **-c** option, none of the configuration files were found from default search locations.

`Could not read file *filename*`

Configuration file reading or closing had failed.

`Internal error`

CMT++ internal sanity checks for its behavior. Assuming your source is correct C/C++ you should not get these. If you are measuring non-preprocessed source code and you are using macros or conditional compilation in some unstructured way, you may end up to this message. Try measuring preprocessed version of the same file.

`License problem: *problem-description*`

There is a problem in your CMT++ license as described in the problem-description, for example "copy protection module not found", "the license has expired", "IP address of this machine (*xx.xx.xx.xx*) is not listed in the COMPUTER field in the configuration file", etc. (These license control routines are used in all Testwell C/C++ test tools)

`Out of memory`

CMT++ detected a short of memory condition.

`Syntax error: unterminated comment`

Comment block starting with /*... ended with EOF.

`There is a problem with software license`

There is a problem with software license.

`Unexpected end of file.`

Unexpected end of file in source file reading.

`No matching '`*`c`*`' between lno and end of file`

On line *lno* there was a starting ‘<’, ‘(‘, or ‘{‘, but it had no matching closing ‘c’ (one of ‘>’, ‘)‘, ‘}’).Your source code is not correct C/C++ or you are measuring non-preprocessed source code and the use of macros or conditional compilation caused CMT++ not to recognize the source correctly.

`Unknown or badly placed option -`*`xxx`*`, type -h to get help`

The option is not known to CMT++ or it is placed after the source file names on the command line.

`Unrecognised input token`

Are you sure that the source code is correct C/C++.

`<Fatal error message related to configuration file handling>`

These messages have error code 2. There are a number of possible messages of this category (used in all Testwell C/C++ tools), like "missing license parameter: *xxx*", "TOOL mismatch", "wrong TOOL version", etc.

# Appendix E. cmt2html Error Messages

The cmt2html error messages are written to *stderr* and they have the following form:

```
cmt2html: error message
```

where the *error message* can be one of:

`Error commandline option: option`
: Bad option when invoking cmt2html.

`Error input line: linenumber`
: The input file seems to be not of the CMT++ report type that cmt2html assumes as input, detected at the given line.

`File filename exists and is not a directory`
: Output directory *filename* could not be created. Rename the blocking file to another name or use another directory name.

`Can not open file filename for writing: op_syst_error_text`
: For some reason this operation failed.

`Can not open file filename for reading: op_syst_error_text`
: For some reason this operation failed.

`Unexpected end of file: inputfilename`
: The input file ended unexpectedly.

# Index