# A Standard For The Measurement Of C Programming Language Attributes

Computer Measurement Laboratory
info@cmlab.biz

In this standard we will develop precise rules for the measurement of 25 program attributes for the C programming language. These rules are intended to be unambiguous and completely reproducible by others. That is the intent of a standard. It is hoped that this standard will serve as a framework for new measures for C program attributes and also as a framework for the development of similar standards for other programming languages.

As a side note, these program metrics are essentially content free. They will tell you little or nothing about a program. They are only data. Measuring the number of operators in a program module will yield a simple number. It is rather like measuring the height of prospective employees. It is a worthless piece of data. If, on the other hand, we were to have learned that taller people make better employees in the long term, then the height measure would be very important. Success in employment is something that we would like to measure directly. However, Nature will never disclose this to us. We can learn from our past employment history. We can measure the success of an employee perhaps in terms of their total annual sales of our Widgets. This new measure of annual sales we will call a criterion measure. It's what we really want to know. We might learn from these data that our employees' height is highly correlated with their annual sales. In other words, height is a good leading indicator of a prospective employee's sales potential. In which case, it makes real good sense to measure the height of a prospective employee. This value is no longer data. It is information.

In the programming world, quite of number of organizations use the Lines of Code, LOC, metric for a measure of productivity. This is a very interesting idea. However, LOC is not defined anywhere. There is no standard for LOC. Different people compute it in different ways. Furthermore, no matter how it is defined LOC is not a good measure of productivity. Some C code is horribly complex. Some is really simple. One hundred lines of really horribly complex C code is not equivalent to 100 lines of straight line well commented C code.

The metrics that are defined in this standard are all distinctly related to software faults. Knowledge of the complexity of a program module also provides real insight into the relative fault burden of the program module. These metrics are also very good leading indicators of programmer productivity. They will be useful, however, only if you have very good and well defined criterion measure data. In other words, you shouldn't start by simply collecting complexity metric data on a program. You should first define at least one criterion measure and then begin to collect very precise and accurate data about the criterion measure. When you have a sufficient history on the criterion measure, then and only then, should you begin to measure the attributes articulated in this standard.

This standard is meant to capture the metrics at the C function level. We will refer to these functions as C program modules. These are the smallest compilable C program units. This standard is carefully defined from the C compiler perspective. It will assume that all compiler directives have been resolved prior to the measurement process. Header files (the .h files) will have been resolved prior to measurement as well.

Each of the will be described in an unambiguous manner so that new measurement tools written to this standard will all produce identical results. That is the whole notion of a standard. These definitions will be organized into seven different measurement groups depending on where these metrics may be obtained in the compilation/linking process.

There are many different C compilers available today. We will formulate a standard for measuring GCC C code.

For measurement purposes a C program module may not contain any compiler directives. To this end, before we measure a C program we must first remove these directives. We will use the C preprocessor CPP to do this. We will use the output of the CPP program as input to our measurement tool.

This standard will reflect the measurement of the output of the CPP program. We will ignore all lines that begin with the '#' character. Will also ignore all lines than contain only the new line character '\n'.

The lexical analysis of a C source program module will resolve the strings of characters on the input stream to a sequence of tokens. These tokens may be classified into the two mutually exclusive classes of operators and operands. There are two distinct ways to enumerate both of the categories of operators and operands. We may count the total times that a particular operator or operand has been used or we chose to enumerate only the first time it occurs. In a language such as C, and operator may be overloaded. That is, it may be called to serve many different duties. Take for example the operator "+". This operator may be used for integer operands or for real operands. Further, it may be used as a unary operator or as a binary operator. Thus, there are two distinct ways in which the unique operator count may be computed. First, we will enumerate the token "+". Next we will parse the C program to determine the context in which the "+" operator has been used and reclassify each new occurrence of this operator as an integer "+" , a floating point "+", etc. This will yield a non-overloaded count of the "+" operators. This also means, however, that we will not be able to tally the total unique non-overloaded operator count until the last, or semantic, phase of the compilation processes.

There are really two distinct classes of tokens that will be used as operators. Some tokens are used by the compiler to control the parsing process and are subsequently discarded, for example, the braces '{' and '}'. Other tokens will result in some kind of runtime action, for example, the adding operators, '+' and '-'. Thus we will want to count these operators separately. One set, the compiler tokens, represents the complexity of the compilation environment. The other set represents the complexity of the execution environment.

There follows, then, a set of metrics that explain a substantial amount of the variation in the criterion measure of software faults.

## Size Metrics

The size metrics have the common characteristics that they simply enumerate various aspects of the program source code. For the most part, these tallies can be derived from the lexical analysis of a program.

### *1. COMM*

This is a count of comment statements. This count must be established for each program module independently. A comment statement begins with the token</*> and ends with the token <*/>. In GCC C a comment statement may also begin with the token <//> and end with the token <\n>.

In a well-designed program there should be no need for comments. The code should map directly from the low-level design document. [c.f. J. C. Munson, *Software Specification and Design: An Engineering Approach*, CRC Press 2006]

### *2. ALOC*

There are actually two measures for lines of code. First there is the Actual Lines of Code (*ALOC*) that the programmer wrote or included in the compilation. This will be enumerated by counting the actual lines of code ***within*** each program module.

This metric will be obtained directly by reading the output of the CPP program. We will ignore all lines that begin with the '#' character. Will also ignore all lines than contain only the new line character '\n'. For each line we will increment our *ALOC* counter until we encounter an end of file.

### *3. RLOC*

There is a second measure of lines of code that reflects the burden added by the programmer's `#include` statements. This is the Real Lines of Code (*RLOC*) metric. This will be obtained by counting the number of statements of declarations, etc. global to the program modules. Then this count will be added to the *ALOC* for each program module.

Again, there will be a *RLOC* metric for each program function module. Thus if there are 30 LOC in a file that are external to all program modules in that file, then the *RLOC* for each program module will be incremented by 30.

### *4. ATOR*

During the lexical analysis phase GCC will scan the input stream to build tokens. These tokens will fall into one of two distinct buckets. They are either operators or operands. The Actual Total Operator count (*ATOR*) will be obtained by counting the number of operator tokens in each program module. By definition, an operator is any token that is not an operand.

Unfortunately, for languages like C a function call is at once an operand and an operator. Thus, during the lexical phase the function call will be recognized correctly as on operand. During the parsing phase we will also recognize this functional call as an operator thus the operator count, *ATOR* must be incremented by one for each such function call.

### *5. ATOD*

This is the Actual Total Operand count (*ATOD*) for each program module. An operand may be one of three distinct tokens. These are as follows: `CONSTANT`, `IDENTIFIER`, or `STRING_LITERAL`.

Again, for languages like C a function call is at once an operand and an operator. Thus, during the lexical phase the function call will be recognized correctly as on operand. During the parsing phase we will also recognize this functional call as an operator thus the operator count, *ATOD* must be incremented by one for each such function call.

### *6. AUOR*

This is the Actual Unique Operator count (*AUOR*). To obtain this value for each program module, a count will be made of the number of distinct (unique) operators that are used in the module.

Function calls are again counted as operators.

### *7. AUOD*

This is a count of the number of Actual Unique Operands (*AUOD*) in each program module. This is, in essence, the number of distinct (unique) identifiers used in each module.

Function calls as also counted as operands.

### *8. RTOR*

This count will reflect the total number of operators in each program module together with the number of operators in the globally defined operators outside of any program module.
The *RTOR* count for each program module will include the operators that are in the global space common to all program modules in a file.

### *9. RTOD*

As above, this count will reflect the incremental number of operators to be added to each program module operand count based on the operands in the externally defined statements.

### *10. RUOR*

This is a count of the number of distinct (unique) operators in a program module including those operators defined in the global space of the file.

### *11. RUOD*

This is a count of the number of distinct (unique) operands in a program module including those operands defined in the global space of the file.

### *12. ACOR*

The Actual Compiler Operators (*ACOR*) constitutes an enumeration of the tokens that are strictly used during the compilation process in each program module. The list of compiler tokens that are compiler operators is shown in Appendix 2.

### *13. AAOR*

The Actual Algorithmic Operators (*AAOR*) is a count of the algorithmic tokens used in each program module that will actually be expressed as executable code. Again this list is shown in Appendix 2.

### *14. RCOR*

As was the case earlier this is a count of the Real Compiler Operators (*RCOR*) that is obtained by adding into the *ACOR* count for each module, the count of the compiler operators

### *15. RAOR*

This is nominally the count of operators inclusive of the external code. There should be no circumstances where there will be operators outside of a program module. Thus this value should be the same as AAOR. We need to verify that this is the case.

### *16. OUOR*

Most of the algorithmic operators in C may be overloaded. That is, they may have different semantic contexts. For example, the '+' sign may be used to add multiple types of integers. It may also be used to at long and short floating point numbers. Thus, in the semantic compiler phase we will need to count the number of unique overloaded operators, one for each type of arithmetic that may be used.

There are two ways that unique operators may be enumerated, overloaded and non-overloaded. *OUOR* is a special variation of the count of the number of unique operators. It represents the count of unique non-overloaded operators. The counting rules are similar to ***AUOR***. In addition, the concept of operator overloading is taken into consideration. ***OUOR*** distinguishes operators that appear to be identical but are intrinsically different because of the operands they work with.

| Example | ***AUOR*** | ***OUOR*** |
|---|---|---|
| {<br>int i1=1,i2=2,i3;<br>float f1=1.1,f2=2.2,f3;<br><br>i3 = i1 + i2;<br>f3 = f1 + f2;<br>} | 8 unique overloaded operators:<br><br>1. {<br>2. int<br>3. =<br>4. ,<br>5. ;<br>6. float<br>7. +<br>8. } | 10 unique operators:<br>1. {<br>2. int<br>3. =int<br>4. ,<br>5. ;<br>6. float<br>7. =float<br>8. +int<br>9. +float<br>10. } |

The following rules apply when computing this metric:

An assignment has always the type of its left-hand expression.

If there is an addition or a multiplication operation of the form:

*operand-1 operator operand-2*

The type of the operator is determined according to the order of precedence. The types of operand-1 and operand-2 are compared and the operation takes the higher type of both within this order. The order of precedence is: double ⇒ float ⇒ long ⇒ int ⇒ short.

Function calls are classified based on the called function's return type.

Unary operators can also be overloaded.

A character is considered an integer except in :

A character assignment

A function declaration

| Example | Considerations |
|---|---|
| char c1, c2;<br>c1 = c1 + c2; | = is a character assignment<br>+ is an integer addition |
| char mod_1()<br>{ }<br><br>main()<br>{<br>mod_1();<br>} | This is a character function call. |

### *17. STMT*

This is an enumeration of executable statements in the C programming language. Unfortunately, there is some ambiguity as to what a statement is in C. We will incorporate the notion of a C expression in our statement count. This is shown in the grammar in Appendix 3.

The token <;> is used to delimit some statements in C. An executable statement is a statement that will change the state of the program at run time. In C, executable statements may stand alone. They may also be embedded in predicate clauses as expressions such as

```
while (j = k < l)
```

In which case, they are delimited by <)>. They may also occur in declarations vis:

```
{
int j = 1, k = 2;
}
```

In this case, the statement `j = 1` is delimited by <,>.

A compound statement delimited by <{> and <}> may or may not contain declarations within its body. If it does contain declarations it is a block. The declarations that differentiate a block from a compound statement will cause the compiler to build an activation stack at run time. Therefore a block is an executable statement in its own right.

The executable statement count must not be modified in the following cases:

within a string constant
within a character constant
within a comment
at the end of a non-executable statement
as a separator in a `for` structure

| Examples | Executable statements |
|---|---|
| `int a;` | 0 |
| `a = b + c;` | 1 |
| `for(i=1; i<50; i++)`<br>`  b[i] = ';';` | 4 |
| `while(a=b<c)`<br>`  a++;` | 2 |

### *18. DECL*

This metric represents the enumeration of non-executable or declarative statement in a program module. In this case we will not distinguish between declarative statements inside a program module and those that are in the global definition space. DECL will include all declarations whether they are in the module or external to it.

Non-executable statements are present in variable declarations, structures, unions, enumerated declarations and type definitions. If a variable is declared, the declaration is considered a non-executable statement. Very simply, non-executable statements will not result in executable code at run time.

The non-executable statement metric must not be modified in the following cases:

within a string constant
within a character constant

within a comment
type definition

| Example | Non-executable statement |
|---|---|
| int i; | 1 |
| int i = 3 ; | 1 |
| typedef int bool; | 1 |
| struct time<br>{<br>  int hour;<br>  int minute;<br>}; | 3 |
| typedef struct time<br>{<br>  int hour;<br>  int minute;<br>} timetype;<br>timetype a, b, *c; | 4 |

## Control Flow Metrics

The metrics in this phase will all relate to the flow graph representation of a program module as defined in the RTL of the GCC compiler. Unfortunately, GCC does not create a distinguished terminal node for its flow graph. We will have to create such a node and then link all `return` statements to it.

A control flowgraph of a program is constructed from a directed graph representation of the program that can be defined as follows:

A directed graph, $G = (N, E, s, t)$, consists of a set of nodes, $N$, a set of edges $E$, a distinguished node $s$, the start node and a distinguished node $t$, the exit node. An edge is an ordered pair of nodes, $(a, b)$.

The in-degree $I(a)$ of node $a$ is the number of entering edges to $a$.

The out-degree $O(a)$ of node $a$ is the number of exiting edges from $a$.

The flowgraph representation of a program, $F = (E', N', s, t)$, is a directed graph that satisfies the following properties.

There is a unique start node $s$ such that $I(s) = 0$.

There is a unique exit node $t$ such that $O(t) = 0$.

All other nodes are processing nodes.

**Processing Node** has one entering edge and one exiting edge. For processing node $a$, $I(a) >= 1$ and $O(a) > = 1$.

If $(a, b)$ is an edge from node $a$ to node $b$, then node $a$ is an immediate predecessor of node $b$ and node $b$ is an immediate successor of node $a$. The set of all immediate predecessors for node $a$ is

denoted as *IP (a)*. The set of all immediate successors for node *b* is denoted as *IS (b)*. No node may have itself as a successor. That is, *a* may not be a member of *IS (a)*.

From this control flow graph representation, two essential control flow primitive metrics emerge:

number of nodes

number of edges

A path *P* in a flowgraph *F* is a sequence of edges $< \overrightarrow{a_1a_2}, \overrightarrow{a_2a_3}, \ldots, \overrightarrow{a_{N-1}a_N} >$ where all $a_i$ *( i = 1,…,N)* are elements of *N'*. *P* is a path from node $a_1$ to node $a_n$. An execution path in *F* is any path *P* from *s* to *t*.

The average length of the paths measured in numbers of edges constitutes a second characteristic of a program. A program that has a large number of relatively short control-flow paths differs greatly in terms of testing or maintenance from one having a few relatively long control-flow paths.

Whether a node lies or not on a cycle relates to the concept of connectedness defined as follows:

A flowgraph *F* is weakly connected if any two nodes *a* and *b* are connected by a sequence of edges.
A flowgraph *F* is strongly connected if each node lies on a cycle.

Any flowgraph might potentially contain weakly connected subsets of nodes that are flowgraphs in their own right. To examine this potential hierarchical structure of the flowgraph representation, the notion of a sub-flowgraph is essential.

A sub-flowgraph *F' = (N'', E'', s', t')* of a flowgraph *F = (N', E', s, t)* is a flowgraph if the out-degree of every node in *F'* is the same as the out-degree of the corresponding node in *F* with the exception of the nodes *s'* and *t'*. All nodes in the sub-flowgraph are weakly connected only to nodes in $N''$.

A sub-flowgraph of *F* is a sub-flowgraph with the property that the cardinality of *N'' > 2* and $F' \neq F$. That is, the sub-flowgraph must contain more nodes than the start and exit nodes and cannot be the same flowgraph.

A flowgraph is an irreducible flowgraph if it contains no proper sub-flowgraph. A flowgraph is a prime flowgraph if it contains no proper sub-flowgraph for which the property *I (s') = I* holds. A prime flowgraph can not be built by sequencing or nesting other flowgraphs and it must contain a single entrance and a single exit structure. The primes are the primitive building blocks of a program control flow.

In the C language the primes relate to the basic control structures of :

```
if (<exp>)
   <stmt>

if (<exp>)
   <stmt-1>
else
   <stmt-2>
```

```
while (<exp>)
   <stmt>

do
   <stmt>
while (<exp>)
```

### *19. NODE*

This is a count of the total number of nodes for a program module flow graph.

The node count represents the number of nodes in the flowgraph representation of the program module. The minimum number of nodes that a module can have is three: the start node, a single processing node and an exit node.

If a module has more than one exit point then a virtual receiving node must be created to meet the condition that there be but one exit node. This virtual node helps to keep the consistency in the flowgraph by ensuring that there is a unique exit node.

The return statement should be considered a processing node (or as a part of a processing node) since it sets the value of what is been returned to the caller function. The processing node must be followed by an edge to the virtual receiving node if multiple return statements are present.

### *20. EDGE*

This is a count of the total number edges in the flow graph representation of a program module.

The Edge metric represents the number of edges in the control flow representation of the program module. The minimum number of edges that a module can have is two in that there must be at least one processing node.

### *21. PATH*

This is an enumeration of the total number of paths through a program module flow graph beginning at the module entry point to a single point of return from the module, the distinguished terminal node.

A path through a flow graph is an ordered set of edges *(s, ... , t)* that begins on the starting node s and ends on the terminal node *t*. A path may contain one or more cycles. Each distinct cycle cannot occur more than once in sequence. That is, the subpath *(a, b, c, a)* is a legal subpath but the subpath *(a, b, c, a, b, c, a)* is not permitted.

The total path set of a node *a* is the set of all paths *(s, a)* that go from the start node to node *a* itself. The cardinality of the set of paths of node *a* is equal to the total path count of the node *a*. Each node singles out a distinct number of paths to the node that begin at the starting node and end with the node itself. The path count of a node is the number of such paths.

The number of distinct execution paths, *Paths*, is computed by systematically decomposing the control flowgraph representation of a program module into constituent prime flowgraphs, a process that systematically eliminates all predicate nodes. The *Path* metric will tally the number of paths that begin at node *s* and end at node *t*.

Cycles are permitted in paths. For each cyclical structure exactly two paths are counted: one that includes the code in the cycle and one that does not. In this sense, each cycle contributes a minimum of two paths to the total path count.

It is a wise idea to set an upper bound on the total path count. A module with *n* predicate clauses in series will generate $2^n$ paths at a minimum. A reasonable upper bound on this count might be 50,000 paths.

### *22. CYCL*

This is a count of the total number of cycles found in a program flow graph. Each cycle will be enumerated exactly once.

A cycle is a collection of strongly connected nodes. From any node in the cycle to any other, there is a path of length one or more, wholly within the cycle. This collection of nodes has a unique entry node. This entry node dominates all nodes in the cycle. A cycle that contains no other cycles is called an inner cycle.

The cycles metric counts the number of cycles in the control flow graph. There are two different types of cycles:

> Iterative loops are those that result from the use of the `for`, `while` and `do/while` structures.
>
> Control loops are created by the backward reference of instructions. If the label object of a `goto` is above the `goto` and there is a possible path from the label to the `goto` statement, it causes a cycle. Thus, each occurrence of such a cycle increases the count of the cycle metric.

### *23. MAXP*

This metric represents the number of edges in the longest path. From the set of available paths for a module, all the paths are evaluated by counting the number of edges in each of them. The greatest value is assigned to this metric. This metric gives an estimate of the maximum path flow complexity that might be obtained when running a module.

## Coupling Metrics

With these metrics we will evaluate coupling complexity. The control flow among the program modules is of concern here. These metrics will have to be obtained from the symbols generated by the linker.

Coupling reflects the degree of relationship that exists between modules. The more coupled two modules are, the more dependent they become from each other. Coupling is an important measurement domain because is closely associated with the impact that a change in a module (or also a fault) might have in the rest of the system.

There are many different attributes that relate specifically to the binding between program modules at run time. These attributes are related to the coupling characteristics of the module structure. For our purposes, we will examine two attributes of this binding process. First, there is the transfer of program control into and out of the program module. Second, there is the flow of data into and out of a program module.

The flow of control among program modules can be represented in a program call graph. This call graph is constructed from a directed graph representation of program modules that can be defined as follows:

> A directed graph, $G = (N, E, s)$, consists of a set of nodes, $N$, a set of edges $E$, a distinguished node $s$, the main program node. An edge is an ordered pair of nodes, $(a, b)$.
>
> There will be an edge $(a, b)$ if program module $a$ can call module $b$.
>
> As was the case for a module flowgraph, the in-degree $I(a)$ of node $a$ is the number of entering edges to $a$.
>
> Similarly, the out-degree $O(a)$ of node $a$ is the number of exiting edges from $a$.

The nodes of a call graph are program modules. The edges of this graph represent calls from module to module and not returns from these calls. Only the modules that are constituents of the program source code library will be represented in the call graph. This call graph will specifically exclude calls to system library functions.

Coupling reflects the degree of relationship that exists between modules. The more tightly coupled two modules are the more dependent they become from each other. Coupling is an important measurement domain because is closely associated with the impact that a change in a module (or also a fault) might have in the rest of the system. There are several program attributes that are associated with coupling. Two of these attributes relate to the binding between a particular module and other program modules. There are two distinct concepts represented in this binding. One is number of modules that can call a given module. The other is the number of calls out of a particular module. These will be represented by the fan-in and fan-out metrics.

### *24. FNIN*

The Fan-In metric (*FNIN*) is a count of the number of modules that call the measured module. For example, if 4 modules call module A, then the Fan-In for A will be 4.

### *25. FNOT*

The Fan-Out metric (*FNOT*) is a count of the number of unique external modules that the measured module can make. This will include the calls into the library functions. We will not tally the total number of calls made to a particular module. Only the first instance of each call will be counted.

# APPENDIX 1

## C Tokens

**Operand Tokens**

Identifier
Constant
String_Literal

**Operator Tokens**

**Keywords**

| | | |
|---|---|---|
| auto | extern | signed |
| break | float | sizeof |
| case | for | static |
| char | goto | struct |
| continue | if | switch |
| const | int | typedef |
| default | long | union |
| do | noalias | unsigned |
| double | register | volatile |
| else | return | void |
| enum | short | while |

**Punctuation**

,
: as label operation
; as statement delimiter

**Blocks**

{ } as block delimiters

**Unary operators:**

& as address operator
* as pointer operator
- as minus operator
+ as plus operator
! as not operator
sizeof( )
(<type>) cast operators
~ as integer complement operator
++ as integer prefix increment operator
++ as short prefix increment operator
++ as long prefix increment operator
+ + as float prefix increment operator
+ + as double prefix increment operator
-- as integer prefix decrement operator
-- as short prefix decrement operator
-- as long prefix decrement operator
-- as float prefix decrement operator
-- as double prefix decrement operator
++ as integer postfix increment operator
++ as short postfix increment operator
++ as long postfix increment operator
++ as float postfix increment operator
++ as double postfix increment operator
-- as integer postfix decrement operator
-- as short postfix decrement operator
-- as long postfix decrement operator
-- as float postfix decrement operator
-- as double postfix decrement operator

**Function calls as operators:**

| | | | | | |
|---|---|---|---|---|---|
| int | function call | float | function call | void | function call |
| short | function call | double | function call | | |
| long | function call | char | function call | | |