C++ Metrics

Basic Count Line Metrics

The basis of C++ count metrics is a set of information stored about each line in a file. A line knows if it is:

- Code
- Comment
- Preprocessor
- Declarative
- Executable
- Inactive

A line metric calculated by the parser is the number of lines that meet the criteria within the scope of a function, class, or file. Here is a list of metrics calculated by the parser and their criteria.

```
int metric = 0;
foreach(line,file){
   if (line meets criteria)
      metric ++;
}
```

Metric	Criteria
AltCountLineBlank	!(Code || Comment || Preprocessor)
AltCountLineCode	Code || Preprocessor
AltCountLineComment	Comment
CountLine	endline – startline + 1
CountLineBlank	!(Code || Comment || Preprocessor || Inactive)
CountLineCode	Code && !Inactive
CountLineCodeDecl	Code && Declarative
CountLineCodeExe	Code && Executable
CountLineComment	Comment && !Inactive
CountLineInactive	Inactive
CountLinePreprocessor	Preprocessor

Basic Count (Not Line) Metrics

There are two other count metrics also calculated by the parser. They are:

- **CountDeclClass:** The number of classes the parser finds
- **CountDeclFunction:** The number of functions the parser finds

Basic Token Metrics

The parser keeps a count of certain tokens that appear within the scope of a function. The number of times a token appears is added to get the following metrics:

```
int metric = countAND +
countOR + …
```

Metric	AND	OR	CASE	CATCH	DO	FOR	IF	?	WHILE	;	SWITCH
Cyclomatic			x	x	x	x	x	x	x		
CyclomaticModified				x	x	x	x	x	x		x
CyclomaticStrict	x	x	x	x	x	x	x	x	x		
CountSemicolon										x	

Tokens also impact the **MaxNesting** metric. The parser keeps track of the current nesting, and the maximum nesting. The current nesting is incremented each time the parser encounters DO, ELSE, FOR, IF, WHILE, and SWITCH, and decremented when it exits those scopes.

Basic Statement Metrics

The statement metrics are completely separate from the line metrics. A line can be executable and declarative, but a statement must be one or the other (or empty). The parser determines whether a statement is executable or declarative and keeps an internal count. For a brief example,

```
for (int i = 0;                 // declarative
   i < 10;                      // executable
   i++)                         // executable
;                               // empty
```

The metrics **CountStmtDecl**, **CountStmtExe**, and **CountStmtEmpty** correspond directly to the parser counts, and **CountStmt** = **CountStmtDecl** + **CountStmtExe** + **CountStmtEmpty.**

Essential and Knot Metrics

The **Knots**, **Essential**, **MinEssentialKnots**, **MaxEssentialKnots**, and **CountPaths** metrics are all calculated by a control flow graph. The control flow graph (which is different than the perl script and never actually displayed in understand) is built during a parse, and given the following tokens:

- PassiveNode (Label)
- BreakNode
- SwitchCaseNode
- ContinueNode
- SwitchDefaultNode
- DoWhileNode
- EndDoWhileNode
- ElseNode
- WhileForNode (While or For)
- EndLoopNode
- GoToNode
- IfNode
- EndIfNode
- ReturnNode
- SwitchNode
- EndSwitchNode

The nodes in the graph are numbered in the order they appear in the code. Knots are where lines cross, and are detected when given two jumps (a to b) and (c to d):

min(a,b) < min(c,d) < max(a,b) and max(c,d) > min (a,b)
or min(a,b) < max(c,d) < max(a,b) and min(c,d) < min (a,b)

The number of Knots is given in the **Knots** metric. **MinEssentialKnots** are the number of knots after the graph has been reduced. **MaxEssentialKnots** is MinEssentialKnots + boundry knots where boundry knots are:

Given a jump from i to j:

i < j and a line starts between i and j and ends at i
i < j and a line starts at j and ends between i and j
i > j and a line starts at j and ends above j or below i
i > j and a line starts below i or above j and ends at i

Essential is the number of extra children in the reduced graph plus one (see code to the right). The **CountPath** metric counts the number of paths through the code.

```
complexity = 1;
for node in graph:
   if ( node.numChildren() > 1)
      complexity = complexity + node.numChildren() - 1
```

****

From here on, metrics could be calculated using the perl or python apis, so code samples are the equivalent code in the python api. For non-api users, the important terms to know are refkindstring and entkindstring. These are essentially text describing entities or references. Commas separate different kinds, and ~ is equivalent to NOT. Below are some sample strings:

"c class ~unknown, c struct ~unkown"	matches any class or struct in the c language that is not unknown
"c file"	matches any c file
"c declare, c define"	matches any reference that is a declare or define reference

****

Average Metrics

Average metrics are the sum of the original metric for each function in scope, divided by the number of functions. For a file, the functions in scope are all the functions defined in that file. For a class, it is all the member functions regardless of file. The code sample is for a class. For a file, instead of ent.refs(…), it would be ent.filerefs("c define", "c function",1).

```
sum = 0
for ref in ent.refs("c define", "c member function",1)
   sum = sum + ref.ent().metric(metriclist)[metricname]
avgmetric = sum / len(ent.refs("c define", "c member function",1))
```

Here are the available average metrics, and the metric they are based on.

Metric	Base Metric
AltAvgLineBlank	AltLineBlank
AltAvgLineCode	AltLineCode
AltAvgLineComment	AltLineComment

AvgCyclomatic	Cyclomatic
AvgCyclomaticModified	CyclomaticModified
AvgCyclomaticStrict	CyclomaticStrict
AvgLine	CountLine
AvgLineBlank	CountLineBlank
AvgLineCode	CountLineCode
AvgLineComment	CountLineComment
AvgEssential	Essential

Average values are not used again to calculate other metrics.

Class Count Metrics

Classes have a bunch of additional count metrics such as CountClassBase and so on. These metrics are all based on references. In general, they are the number of references that have the given kind of entity (entkindstring) and kind of reference (refkindstring). Not all metrics have both an entkindstring and a refkindstring.

```
metric = len( class.refs(refkindstring, entkindstring,True) )
```

Metric	RefKindString	EntKindString
CountClassBase	“c base”	
CountDeclClassMethod	“c declare, c define”	“c member function static”
CountDeclClassVariable	“c declare, c define”	“c member object static”
CountDeclInstanceMethod	“c declare, c define”	“c member function ~static”
CountDeclInstanceVariable	“c declare, c define”	“c member object ~static”
CountDeclInstanceVariablePrivate	“c declare, c define”	“c private member object ~static”
CountDeclInstanceVariableProtected	“c declare, c define”	“c protected member object ~static”
CountDeclInstanceVariablePublic	“c declare, c define”	“c public member object ~static”
CountDeclMethod	"c declare,c define"	,"c member function ~implicit"
CountDeclMethodConst	“c declare, c define”	“c member function const ~implicit”
CountDeclMethodPrivate	“c declare, c define”	“c member private function ~implicit”
CountDeclMethodProtected	“c declare, c define”	"c member protected function ~implicit"
CountDeclMethodPublic	“c declare, c define”	"c member public

	function ~implicit"
CountClassDerived	"c derive"

A few exceptions:

CountDeclMethodAll- This is really similar to the above metrics, except it gets the references from all the base classes (recursively), as well as the particular class being referenced. It uses the refkindstring "c declare ~using, c define" and the entkindstring "c member function ~implicit"

```
#CountDeclMethodAll
refstr = "c declare ~using,c define"
entstr = "c member function ~implicit"
bases = allbases()
metric = 0
for base in bases:
   metric = metric + len(base.refs(refstr,entstr))
```

CountDeclMethodFriend- This metric is almost the same as using the refkindstring "c friend" with entkindstring "c function" except that if a class is a friend, then all of that classes methods are also added to the metric.

```
#CountDeclMethodFriend
metric = 0
methods = ["CountDeclMethod"]
classstr = "c class, c struct, c union"
for ref in class.refs("c friend","",1):
   if(ref.ent().kind().check("c function")):
      metric = metric + 1
   elif (ref.ent().kind().check(classtr)):
      metric = metric + ref.ent().metric((methods))[methods]
```

CountClassCoupled- This metric counts the number of classes that this class interacts with. It is more complicated to calculate, but the idea is to take the class, all of it's functions, and all of the entities used in the function and grab all the references. Then, make a unique list of any classes that are the referenced excluding base classes and nested classes. A class is referenced if it, it's methods, or any of it's entities are used. The metric is equal to the total number of unique classes referenced.

Max Metrics

MaxCyclomatic, MaxCyclomaticModified, and **MaxCyclomaticStrict** are all fairly straightforward. For a file or class, simply take all the functions and find the one with the maximum Cyclomatic, CyclomaticModified, or CyclomaticStrict and return that value. However, the remaining max metrics ask for a little more explanation.

```
refs
if ent.kind().check("c file"):
  refs = ent.filerefs("c define", "c function",1)
else: #class
   refs = ent.refs("c define", "c member function",1)

max = 0
for ref in refs:
   val = ref.ent().metric(["metric"])["metric"]
   if ( val > max) :
      max = val
```

MaxNesting- For functions, this is explained under Basic Token Metrics. For files, this is calculated the same way MaxCyclomatic is.

MaxInheritanceTree- This calculates the maximum number of classes between this class and the root class of the inheritance tree. It is calculated by recursively asking for the MaxInheritanceTree of the base classes and adding one. The root base class has a MaxInheritanceTree value of 0.

Sum Metrics

Sum metrics are all straightforward. Given a file or class, add together the metric value for each of the functions. The sum metrics are:

- **SumCyclomatic**
- **SumCyclomaticModified**
- **SumCyclomatiStrict**
- **SumEssential**

```
refs
if ent.kind().check("c file"):
  refs = ent.filerefs("c define", "c function",1)
else: #class
    refs = ent.refs("c define", "c member function",1)

sum = 0
for ref in refs:
    sum = sum + ref.ent().metric(["metric"])["metric"]
```

Other Metrics

CountInput is like many of the class count metrics because it is the number of references that match the refkindstring "c use ~ptr ~inactive, c callby ~inactive" and the entkindstring "c global object, c local object, c member object, c parameter, c function". However, it does some additional filtering to ensure that recursive calls are removed, and that the only local objects kept are static members of classes.

CountOutput is the same as CountInput, except the refkindstring is "c set, c modify, c call ~inactive" and there is an additional filter that ensures parameters are only counted if they are references or pointers.

RatioCommentToCode: This is just CountLineComment / CountLineCode

PercentLackOfCohesion: This is 1 – Average(class functions that use instance variable / total functions in class).