Writing Cppcheck rules

Part 2 - The Cppcheck data representation

Daniel Marjamäki, Cppcheck

2010

Introduction

In this article I will discuss the data representation that Cppcheck uses.

The data representation that Cppcheck uses is specifically designed for static analysis. It is not intended to be generic and useful for other tasks.

See the data

There are two ways to look at the data representation at runtime.

Using --rule=.+ is one way. All tokens are written on a line:

```
 int a ; int b ;
```

Using --debug is another way. The tokens are line separated in the same way as the original code:

```
1: int a@1 ;
2: int b@2 ;
```

In the --debug output there are "@1" and "@2" shown. These are the variable ids (Cppcheck gives each variable a unique id). You can ignore these if you only plan to write rules with regular expressions, you can't use variable ids with regular expressions.

In general, I will use the `--rule=.+` output in this article because it is more compact.

Some of the simplifications

The data is simplified in many ways.

Preprocessing

The Cppcheck data is preprocessed. There are no comments, #define, #include, etc.

Original source code:

```
#define SIZE 123
char a[SIZE];
```

The Cppcheck data for that is:

```
 char a [ 123 ] ;
```

typedef

The typedefs are simplified.

```
typedef char s8;
s8 x;
```

The Cppcheck data for that is:

```
 ; char x ;
```

Calculations

Calculations are simplified.

```
int a[10 + 4];
```

The Cppcheck data for that is:

```
 int a [ 14 ] ;
```

Variables

Variable declarations

Variable declarations are simplified. Only one variable can be declared at a time. The initialization is also broken out into a separate statement.

```
int *a=0, b=2;
```

The Cppcheck data for that is:

```
int * a ; a = 0 ; int b ; b = 2 ;
```

This is even done in the global scope. Even though that is invalid in C/C++.

Known variable values

Known variable values are simplified.

```
void f()
{
    int x = 0;
    x++;
    array[x + 2] = 0;
}
```

The `--debug` output for that is:

```
1: void f ( )
2: {
3: ; ;
4: ;
5: array [ 3 ] = 0 ;
6: }
```

The variable x is removed because it is not used after the simplification. It is therefore redundant.

The "known values" doesn't have to be numeric. Variable aliases, pointer aliases, strings, etc should be handled too.

Example code:

```
void f()
{
    char *a = strdup("hello");
    char *b = a;
    free(b);
}
```

The `--debug` output for that is:

```
1: void f ( )
2: {
3: char * a@1 ; a@1 = strdup ( "hello" ) ;
4: ; ;
5: free ( a@1 ) ;
6: }
```

if/for/while

Braces in if/for/while-body

Cppcheck makes sure that there are always braces in if/for/while bodies.

```
    if (x)
        f1();
```

The Cppcheck data for that is:

```
 if ( x ) { f1 ( ) ; }
```

No else if

The simplified data representation doesn't have "else if".

```
void f(int x)
{
    if (x == 1)
        f1();
    else if (x == 2)
        f2();
}
```

The `--debug` output:

```
1: void f ( int x@1 )
2: {
3: if ( x@1 == 1 ) {
4: f1 ( ) ; }
5: else { if ( x@1 == 2 ) {
6: f2 ( ) ; } }
7: }
```

Condition is always true / false

Conditions that are always true / false are simplified.

```
void f()
{
    if (true) {
        f1();
    }
}
```

The Cppcheck data is:

```
 void f ( ) { { f1 ( ) ; } }
```

Another example:

```
void f()
{
    if (false) {
        f1();
    }
}
```

The debug output:

```
 void f ( ) { }
```

Assignments

Assignments within conditions are broken out from the condition.

```
void f()
{
    int x;
    if ((x = f1()) == 12) {
        f2();
    }
}
```

The `x=f1()` is broken out. The `--debug` output:

```
1: void f ( )
2: {
3: int x@1 ;
4: x@1 = f1 ( ) ; if ( x@1 == 12 ) {
5: f2 ( ) ;
6: }
7: }
```

Replacing the "if" with "while" in the above example:

```
void f()
{
    int x;
    while ((x = f1()) == 12) {
        f2();
    }
}
```

The `x=f1()` is broken out twice. The `--debug` output:

```
1: void f ( )
2: {
3: int x@1 ;
4: x@1 = f1 ( ) ; while ( x@1 == 12 ) {
5: f2 ( ) ; x@1 = f1 ( ) ;
5:
6: }
7: }
```

Comparison with >

Comparisons are simplified. The two conditions in this example are logically the same:

```
void f()
{
    if (x < 2);
    if (2 > x);
}
```

Cppcheck data doesn't use > for comparisons. It is converted into < instead. In the Cppcheck data there is no difference for `2>x` and `x<2`.

```
1:
2: void f ( )
3: {
4: if ( x < 2 ) { ; }
5: if ( x < 2 ) { ; }
6: }
```

A similar conversion happens when >= is used.

if (x) and if (!x)

If possible a condition will be reduced to x or !x. Here is an example code:

```
void f()
{
    if (!x);
    if (NULL == x);
    if (x == 0);

    if (x);
    if (NULL != x);
    if (x != 0);
}
```

The `--debug` output is:

```
1: void f ( )
2: {
3: if ( ! x ) { ; }
4: if ( ! x ) { ; }
5: if ( ! x ) { ; }
```

```
6:
7: if ( x ) { ; }
8: if ( x ) { ; }
9: if ( x ) { ; }
10: }
```