Writing Cppcheck rules

Part 1 - Getting started

Daniel Marjamäki, Cppcheck

2010

Introduction

This is a short and simple guide that describes how rules are written for Cppcheck.

The patterns are defined with regular expressions. It is required that you know how regular expressions work.

Data representation of the source code

The data used by the rules are not the raw source code. `Cppcheck` will read the source code and process it before the rules are used.

Cppcheck is designed to find bugs and dangerous code. Stylistic information (such as indentation, comments, etc) are filtered out at an early state. You don't need to worry about such stylistic information when you write rules.

Between each token in the code there is always a space. For instance the raw code "1+f()" is processed into "1 + f ( )".

The code is simplified in many ways.

Creating a simple rule

When creating a rule there are two steps:

1. Create the regular expression
2. Create a XML based rule file

Step 1 - Creating the regular expression

Cppcheck uses the `PCRE` library to handle regular expressions. `PCRE` stands for "Perl Compatible Regular Expressions". The homepage for `PCRE` is `http://www.pcre.org`.

Let's create a regular expression that checks for code such as:

```
if (p)
    free(p);
```

For such code the condition is often redundant (on most implementations it is valid to free a NULL pointer).

The regular expression must be written for the simplified code. To see what the simplified code looks like you can create a source file with the code:

```
void f() {
```

```
    if (p)
        free(p);
}
```

Save that code as `dealloc.cpp` and then use `cppcheck --rule=".+" dealloc.cpp`:

```
$ ./cppcheck --rule=".+" dealloc.cpp
Checking dealloc.cpp...
[dealloc.cpp:1]: (style) found ' void f ( ) { if ( p ) { free ( p ) ; } }'
```

The regular expression `.+` matches everything and the matching text is shown on the screen.

From that output we can see that the simplified code is:

```
 void f ( ) { if ( p ) { free ( p ) ; } }
```

Now that we know how the simplified code looks. We can create a regular expression that matches it properly:

```
$ cppcheck --rule="if \( p \) { free \( p \) ; }" dealloc.cpp
Checking dealloc.cpp...
[dealloc.cpp:2]: (style) found 'if ( p ) { free ( p ) ; }'
```

Step 2 - Create rule file

A rule file is a simple XML file that contains:

- a pattern to search for
- an error message that is reported when pattern is found

Here is a simple example:

```
<?xml version="1.0"?>
<rule version="1">
  <pattern>if \( p \) { free \( p \) ; }</pattern>
  <message>
    <id>redundantCondition</id>
    <severity>style</severity>
    <summary>Redundant condition. It is valid to free a NULL pointer.</summary>
  </message>
</rule>
```

If you save that xml data in `dealloc.rule` you can test this rule:

```
$ cppcheck --rule-file=dealloc.rule dealloc.cpp
Checking dealloc.cpp...
[dealloc.cpp:2]: (style) Redundant condition. It is valid to free a NULL pointer.
```