Writing Cppcheck rules

Part 3 - Introduction to writing rules with C++

Daniel Marjamäki, Cppcheck

2011

Introduction

The goal for this article is to introduce how `Cppcheck` rules are written with C++. With C++ it is possible to write more complex rules than is possible with regular expressions.

Basics

A C++ rule is written in a C++ function.

Rules are organized into Check classes. For instance there is a class with the name `CheckStl` that contains various stl rules. The `CheckOther` can always be used if no other class suits you.

When you have added your rule you must recompile Cppcheck before you can test it.

Division by zero

This simple regular expression will check for division by zero:

```
cppcheck --rule="/ 0"
```

Here is the corresponding C++ check:

```
// Detect division by zero
void CheckOther::divisionByZero()
{
    // Loop through all tokens
    for (const Token *tok = _tokenizer->tokens(); tok; tok = tok->next())
    {
        // check if there is a division by zero
        if (Token::Match(tok, "/ 0"))
        {
            // report error
            divisionByZeroError(tok);
        }
    }
}

// Report error
void CheckOther::divisionByZeroError()
{
    reportError(tok,                  // location
                Severity::error,      // severity
                "divisionByZero",     // id
                "Division by zero");  // message
}
```

The `Token::Match` matches tokens against expressions. A few rules about Token::Match expressions are:

- tokens are either completely matched or not matched at all. The token "abc" is not matched by "ab".
- Spaces are used as separators.
- With normal regular expressions there are special meanings for + * ? ( ). These are just normal characters in `Token::Match` patterns.

Condition before deallocation

In the first `Writing rules` article I described a rule that looks for redundant conditions. Here is the regular expression that was shown:

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

The corresponding `Token::Match` expression is:

```
if ( %var% ) { free ( %var% ) ; }
```

The `%var%` pattern match any variable name. Here is a C++ function:

```
// Find redundant condition before deallocation
void CheckOther::dealloc()
{
    // Loop through all tokens
    for (const Token *tok = _tokenizer->tokens(); tok; tok = tok->next())
    {
        // Is there a condition and a deallocation?
        if (Token::Match(tok, "if ( %var% ) { free ( %var% ) ; }"))
        {
            // Get variable name used in condition:
            const std::string varname1 = tok->strAt(2);

            // Get variable name used in deallocation:
            const std::string varname2 = tok->strAt(7);

            // Is the same variable used?
            if (varname1 == varname2)
            {
                // report warning
                deallocWarning(tok);
            }
        }
    }
}

// Report warning
void CheckOther::deallocWarning()
{
    reportError(tok,                      // location
                Severity::warning,        // severity
                "dealloc",                // id
                "Redundant condition");   // message
```

```
}
```

The strAt function is used to fetch strings from the token list. The parameter specifies the token offset. The result for "tok->tokAt(1)" is the same as for "tok->next()".

Validate function parameters

Sometimes it is known that a function can't handle certain parameters. Here is an example rule that checks that the parameters for strtol or strtoul are valid:

```
//---------------------------------------------------------------------------
// strtol(str, 0, radix)  <- radix must be 0 or 2-36
//---------------------------------------------------------------------------

void CheckOther::invalidFunctionUsage()
{
    // Loop through all tokens
    for (const Token *tok = _tokenizer->tokens(); tok; tok = tok->next())
    {
        // Is there a function call for strtol or strtoul?
        if (!Token::Match(tok, "strtol|strtoul ("))
            continue;

        // Locate the third parameter of the function call..

        // Counter that counts the parameters.
        int param = 1;

        // Scan the function call tokens. The "tok->tokAt(2)" returns
        // the token after the "("
        for (const Token *tok2 = tok->tokAt(2); tok2; tok2 = tok2->next())
        {
            // If a "(" is found then jump to the corresponding ")"
            if (tok2->str() == "(")
                tok2 = tok2->link();

            // End of function call.
            else if (tok2->str() == ")")
                break;

            // Found a ",". increment param counter
            else if (tok2->str() == ",")
            {
                ++param;

                // If the param is 3 then check if the parameter is valid
                if (param == 3)
                {
                    if (Token::Match(tok2, ", %num% )"))
                    {
                        // convert next token into a number
                        MathLib::bigint radix;
                        radix = MathLib::toLongNumber(tok2->strAt(1));
```

```
                        // invalid radix?
                        if (!(radix == 0 || (radix >= 2 && radix <= 36)))
                        {
                            dangerousUsageStrtolError(tok2);
                        }
                    }
                    break;
                }
            }
        }
    }
}

void CheckOther::dangerousUsageStrtolError(const Token *tok)
{
    reportError(tok,                            // location
                Severity::error,                // severity
                "dangerousUsageStrtol",         // id
                "Invalid radix");               // message
}
```

The link() member function is used to find the corresponding ( ) [ ] or { } token.

The inner loop is not necessary if you just want to get the last parameter. This code will check if the last parameter is numerical..

```
..
        // Is there a function call?
        if (!Token::Match(tok, "do_something ("))
            continue;

        if (Token::Match(tok->next()->link()->tokAt(-2), "(|, %num% )"))
            ...
```

The pattern `(|,` can also be written as `[(,]`.