Cppcheck Design

Author: Daniel Marjamäki

Date: 2014-01-21

Introduction

This is a brief overview of the internal data in Cppcheck used by checks.

Preprocessor

The source files are preprocessed. The checkers don't have access to neither the unpreprocessed raw code nor the code that comes directly from the preprocessor.

Token list

The preprocessed code is tokenized. If the source code is "abc=ab+c;" then the token list will have 6 tokens: abc , = , ab , + , c , ;

The checkers have access to the token list. Checks often loop through tokens. Here is a loop that loops through all tokens in the entire translation unit:

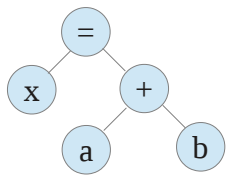
```
// Loop through all tokens
for (const Token *tok = _tokenizer->tokens(); tok; tok = tok->next()) {
    ...
}
```

Syntax tree

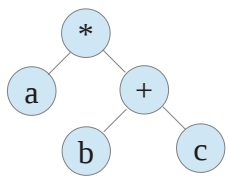
Every statement has a syntax tree. The tokens in the token list are used as nodes in the syntax tree. The tokens in the token list has member functions astParent(), astOperand1() and astOperand2().

Some examples:

```
x = a + b;
```

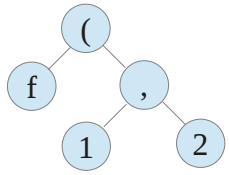



```
a * (b + c)
```



Parentheses are typically not included in the syntax tree. However the start parenthesis is included in the syntax tree for function calls and for casts:

```
f(1,2)
```



Symbol database

A database that contains information about types,functions,etc in the source code. Checks can for instance use this when a type/variable/function/etc is seen in the token list and the check wants to get information about this type/variable/function/etc.

Here is an example code that shows how a checker can see if a function that is called is static:

```
// Is there a function call here?
if (Token::Match(token, "%type% (")) {
    // Get function information about the called function
    const Function *function = token->function();

    // Check if function is static..
    if (function && function->isStatic()) {
        ...
    }
}
```

Here is an example code that shows how a checker can investigate a variable:

```
// Addition of variable with something else
if (Token::Match(token, "%var% +")) {
    // Get variable information
    const Variable *var = token->variable();

    // Is variable a pointer?
    if (var && var->isPointer()) {
        .. yes variable is a pointer ..
    }
}
```

Value flow

Before any checkers are executed, generic context-sensitive value flow analysis is made. The possible variable values will be tracked (backwards, forwards, and into function calls) in the code. And the values will be stored in the corresponding tokens. After this, each token has a list of possible values.

For example, here is source code:

```
void f(int x) {
    int a = 1 + x * x;
}

void otherfunction() {
    f(2);
    f(4);
}
```

The ValueFlow set the following values on the tokens in the function f:

```
1:{1}
+:{5,17}
x:{2,4}
*:{4,16}
x:{2,4}
```

As can be seen, the values of operators will be the possible results of those operators.

Here is a simple example that shows how a ”division by zero” check can be written:

```
// We have a token, is it a division operator?
if (token->str() == ”/”)) {
    // Get RHS operand of division operator
    const Token *rhs = token->astOperand2();

    // Get ”0” value of RHS operand if it exists
    const ValueFlow::Value *val0 = rhs->getValue(0);

    // Is there a ”0” value for RHS?
    if (val0 != NULL) {
        .. yes there is a ”0” value of RHS ..
    }
}
```

Library

The library contains the information the user provide in `--library` configuration files. These configuration files contain properties of functions. Such as if a given function returns or not, if it allocates memory, what arguments is invalid, etc. This type of information should not be hard coded in checks but should be taken from the library instead.