Cppcheck 1.40

Cppcheck 1.40

Table of Contents

Chapter 1. Introduction

Cppcheck is an analysis tool for C/C++ code. Unlike C/C++ compilers and many other analysis tools, it doesn't detect syntax errors. Cppcheck only detects the types of bugs that the compilers normally fail to detect. The goal is no false positives.

Supported code and platforms:

- You can check non-standard code that includes various compiler extensions, inline assembly code, etc.
- Cppcheck should be compilable by any C++ compiler that handles the latest C++ standard.
- Cppcheck should work on any platform that has sufficient cpu and memory.

Accuracy

Please understand that there are limits of Cppcheck. Cppcheck is rarely wrong about reported errors. But there are many bugs that it doesn't detect.

You will find more bugs in your software by testing your software carefully, than by using Cppcheck. You will find more bugs in your software by instrumenting your software, than by using Cppcheck. But Cppcheck can still detect some of the bugs that you miss when testing and instrumenting your software.

Chapter 2. Getting started

First test

Here is a simple code

```
int main()
{
    char a[10];
    a[10] = 0;
    return 0;
}
```

If you save that into `file1.c` and execute:

```
cppcheck file1.c
```

The output from cppcheck will then be:

```
Checking file1.c...
[file1.c:4]: (error) Array index out of bounds
```

Checking all files in a folder

Normally a program has many sourcefiles. And you want to check them all. Cppcheck can check all sourcefiles in a directory:

```
cppcheck path
```

If "path" is a folder then cppcheck will check all sourcefiles in this folder.

```
Checking path/file1.cpp...
1/2 files checked 50% done
Checking path/file2.cpp...
2/2 files checked 100% done
```

Possible errors

By default, an error is only reported when `Cppcheck` is sure there is an error. When `--enable=possibleError` is given issues will also be reported when `Cppcheck` is unsure.

The `--enable=possibleError` flag is useful but makes `Cppcheck` more unreliable, you will probably get false positives.

Here is a simple code example:

```
void f()
{
    Fred *f = new Fred;
}
```

Execute this command:

```
cppcheck --enable=possibleError file1.cpp
```

The output from Cppcheck:

```
[file1.cpp:4]: (possible error) Memory leak: fred
```

The "possible" means that the reported message may be wrong (if Fred has automatic deallocation it is not a memory leak).

Stylistic issues

By default Cppcheck will only check for bugs. There are also a few checks for stylistic issues.

Here is a simple code example:

```
void f(int x)
{
    int i;
    if (x == 0)
    {
        i = 0;
    }
}
```

To enable stylistic checks, use the --style flag:

```
cppcheck --enable=style file1.c
```

The reported error is:

```
[file3.c:3]: (style) The scope of the variable i can be limited
```

Saving results in file

Many times you will want to save the results in a file. You can use the normal shell redirection for piping error output to a file.

```
cppcheck file1.c 2> err.txt
```

Unused functions

This check will try to find unused functions. It is best to use this when the whole program is checked, so that all usages is seen by cppcheck.

```
cppcheck --enable=unusedFunctions path
```

Enable all checks

To enable all checks your can use the `--enable=all` flag:

```
cppcheck --enable=all path
```

Multithreaded checking

To use 4 threads to check the files in a folder:

```
cppcheck -j 4 path
```

Chapter 3. XML output

Cppcheck can generate the output in XML format.

Use the --xml flag when you execute cppcheck:

```
cppcheck --xml file1.cpp
```

The xml format is:

```
<?xml version="1.0"?>
<results>
  <error file="file1.cpp" line="123" id="someError"
               severity="error" msg="some error text"/>
</results>
```

Attributes:

file	filename. Both relative and absolute paths are possible
line	a number
id	id of error. These are always valid symbolnames.
severity	one of: error / possible error / style / possible style
msg	the error message

Chapter 4. Reformatting the output

If you want to reformat the output so it looks different you can use templates.

To get Visual Studio compatible output you can use "--template vs":

```
cppcheck --template vs gui/test.cpp
```

This output will look like this:

```
Checking gui/test.cpp...
gui/test.cpp(31): error: Memory leak: b
gui/test.cpp(16): error: Mismatching allocation and deallocation: k
```

To get gcc compatible output you can use "--template gcc":

```
cppcheck --template gcc gui/test.cpp
```

The output will look like this:

```
Checking gui/test.cpp...
gui/test.cpp:31: error: Memory leak: b
gui/test.cpp:16: error: Mismatching allocation and deallocation: k
```

You can write your own pattern (for example a comma-separated format):

```
cppcheck --template "{file},{line},{severity},{id},{message}" gui/test.cpp
```

The output will look like this:

```
Checking gui/test.cpp...
gui/test.cpp,31,error,memleak,Memory leak: b
gui/test.cpp,16,error,mismatchAllocDealloc,Mismatching allocation and deallocation
```

Chapter 5. Suppressions

If you want to filter out certain errors you can suppress these. First you need to create a suppressions file. The format is:

```
[error id]:[filename]
[error id]:[filename2]
[error id]
```

The `error id` is the id that you want to suppress. The easiest way to get it is to use the `--xml` command line flag. Copy and paste the `id` string from the xml output.

Here is an example:

```
memleak:file1.cpp
exceptNew:file1.cpp
uninitvar
```

You can then use the suppressions file:

```
cppcheck --suppressions suppressions.txt src/
```

Chapter 6. Leaks

Looking for memory leaks and resource leaks is a key feature of Cppcheck. Cppcheck can detect many common mistakes by default. But through some tweaking you can both increase the capabilities and also reduce the amount of false positives.

Automatic deallocation

A common cause of false positives is when there is automatic deallocation. Here is an example:

```
void Form1::foo()
{
    QPushButton *pb = new QPushButton("OK", this);
}
```

Cppcheck can't see where the deallocation is when you have such code.

If you execute:

```
cppcheck --enable=possibleError file1.cpp
```

The result will be:

```
[file1.cpp:4]: (possible error) Memory leak: pb
```

The "possible" in the error message means that the message may be a false positive.

To avoid such false positives, create a textfile and write the names of the automaticly deallocated classes.

```
QLabel
QPushButton
```

Then execute cppcheck with the `--auto-dealloc` option:

```
cppcheck --auto-dealloc qt.lst --enable=possibleError file1.cpp
```

Userdefined allocation/deallocation functions

`Cppcheck` understands many common allocation and deallocation functions. But not all.

Here is example code that might leak memory or resources:

```
void foo(int x)
{
    void *f = CreateFred();
    if (x == 1)
        return;
    DestroyFred(f);
}
```

If you analyse that with Cppcheck it won't find any leaks:

```
cppcheck --enable=possibleError fred1.cpp
```

You can add some custom leaks checking by providing simple implementations for the allocation and deallocation functions. Write this in a separate file:

```
void *CreateFred()
{
    return malloc(100);
}

void DestroyFred(void *p)
{
    free(p);
}
```

When Cppcheck see this it understands that CreateFred will return allocated memory and that DestroyFred will deallocate memory.

Now, execute `Cppcheck` this way:

```
cppcheck --append=fred.cpp fred1.cpp
```

The output from cppcheck is:

```
Checking fred1.cpp...
[fred1.cpp:5]: (error) Memory leak: f
```

Chapter 7. Exception safety

Cppcheck has a few checks that ensure that you don't break the basic guarantee of exception safety. It doesn't have any checks for the strong guarantee yet.

Example:

```
Fred::Fred() : a(new int[20]), b(new int[20])
{
}
```

By default cppcheck will not detect any problems in that code.

To enable the exception safety checking you can use `--enable`:

```
cppcheck --enable=exceptNew --enable=exceptRealloc fred.cpp
```

The output will be:

```
[fred.cpp:3]: (style) Upon exception there is memory leak: a
```

If an exception occurs when `b` is allocated, `a` will leak.

Here is another example:

```
int *p;

int a(int sz)
{
    delete [] p;
    if (sz <= 0)
        throw std::runtime_error("size <= 0");
    p = new int[sz];
}
```

Check that with Cppcheck:

```
cppcheck --enable=exceptNew --enable=exceptRealloc except2.cpp
```

The output from Cppcheck is:

```
[except2.cpp:7]: (error) Throwing exception in invalid state, p points at dealloca
```

Chapter 8. html report

You can convert the xml output from cppcheck into a html report. You'll need python and the pygments module (`http://pygments.org/`) for this to work. In the Cppcheck source tree there is a folder "htmlreport" that contains a script that transforms a Cppcheck xml file into html output.

This command generates the help screen:

```
htmlreport/cppcheck-htmlreport -h
```

The output screen says:

```
Usage: cppcheck-htmlreport [options]

Options:
  -h, --help      show this help message and exit
  --file=FILE     The cppcheck xml output file to read defects from.
                  Default is reading from stdin.
  --report-dir=REPORT_DIR
                  The directory where the html report content is written.
  --source-dir=SOURCE_DIR
                  Base directory where source code files can be found.
```

An example usage:

```
./cppcheck gui/test.cpp --xml 2> err.xml
htmlreport/cppcheck-htmlreport --file=err.xml --report-dir=test1 --source-dir=.
```