

## Cppcheck 1.39

# 目次

|                           |    |
|---------------------------|----|
| 1. はじめに .....             | 3  |
| 2. さあ始めよう！ .....          | 4  |
| 最初のテスト .....              | 4  |
| フォルダ内のすべてのファイルをチェック ..... | 4  |
| 確信のないエラー報告 .....          | 5  |
| スタイルの問題 .....             | 5  |
| 結果をファイルに保存 .....          | 6  |
| 未使用関数検出 .....             | 6  |
| とにかく全てのチェックを行う.....       | 6  |
| 並列チェック.....               | 6  |
| 3. XML 出力 .....           | 7  |
| 4. 出力の整形 .....            | 8  |
| 5. エラー無視 .....            | 9  |
| 6. メモリリーク .....           | 10 |
| 自動解放するヤツら .....           | 10 |
| ユーザー定義のアロケート関数 .....      | 10 |
| 7. 例外安全 .....             | 12 |

# Chapter 1. はじめに

Cppcheck は C/C++の静的解析ツールです。

C/C++コンパイラや、その他多くの解析ツールとは違い、Cppcheckは構文エラーを検出しません。

しかし、通常コンパイラが検出しないタイプのバグの種を検出します。

目標は、0%の誤認識としています。

## サポートしているコード：

- ・ 標準ライブラリ以外のライブラリコードを含むコンパイラのコードもチェックできます。
- ・ インラインアセンブラを含むコードをチェックできます。

## サポートしているプラットフォーム：

- ・ Cppcheckをソースコードからコンパイルする場合、Cppcheckは最新のC++規格を処理するどんなC++コンパイラでもコンパイル可能であるはずです。たぶん。
- ・ CppcheckはそれなりのCPUとそれなりのメモリを積んでいるあらゆるプラットフォームで実行が可能です。たぶん。

## 精度について：

Cppcheckにも限界があります。

Cppcheckはめったに間違った検出報告をしません。

しかし、そもそもCppcheckが検出しない多くのバグの種があります。

あなたは、Cppcheckではない何か別の小洒落たツールを使ってあなたのコードを検査することによって、より多くのバグの種を見つけるでしょう。

しかし、Cppcheckがそれらのツールが検出しない、いくつかのバグの種を検出し、あなたの助けになるであろうことも紛れもない事実です。

## Chapter 2. さあはじめよう！

### 最初のテスト

サンプルコード

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

上記のコードをfile1.cという名前で保存してcppcheckを実行します：

```
cppcheck file1.c
```

出力結果：

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

### フォルダ以下の全てのファイルをチェック

通常、プログラムには、多くのソースファイルがあります。そして、ほとんどのCppcheckユーザーは、それら全てをチェックしたいと思うでしょう。

喜んでください。Cppcheckはディレクトリのすべてのソースファイルをチェックできます！

```
cppcheck path
```

*path* にフォルダを指定した場合 cppcheckはフォルダ内の全てのファイルをチェックします。

出力結果：

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

## 確信のないエラー報告

デフォルトの設定ではCppcheckが明らかな誤りを検出した場合のみエラーレポートが表示されます。

--enable=possibleError という設定をした場合のみ、cppcheckは不確かなレポートも出力します。

しかし、これによりcppcheckの精度は落ち、頼りなくなってしまうでしょう。

あなたはたぶん誤った検出報告を得ることになります。

### サンプルコード

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

上記のコードに以下のコマンドでcppcheckを実行します。

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

### 出力結果

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

“possible”はこのレポートが間違っているかもしれないということを意味しています

(Fredがスマートポインタだったり、デストラクタでdelete thisしてるかもしれないという可能性があるので必ずメモリリークしているとは言いきれません)

## スタイルの問題

デフォルトの設定ではCppcheckは忌まわしきバグの種がないかどうかチェックするだけです

しかしCppcheckが本気になれば、コーディングのスタイルの問題もチェックすることができます。

### サンプルコード

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

```
        i = 0;
    }
}
```

スタイルをチェックしたい場合は`-style`フラグを指定します

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

出力結果:

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

(変数 `i` はもっと狭いスコープでインスタンスできるだろーが! の意)

### 結果をファイルに保存

あなたはcppcheckの出力をファイルに保存したいと思うでしょう。

通常のシェルのリダイレクトを使って結果をファイルに保存できます。

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

### 未使用関数検出

未使用関数の検出ができます。

プログラム全体をチェックする時、この機能を使用するとよいでしょう

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

### とにかく全てのチェックを行う

`--enable=all` を指定すれば全てのチェックを有効にできます。

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

### 並列チェック

例えば、4つのスレッドを使ってチェックする場合は以下のコマンドを使用します。

```
cppcheck -j 4 path
```

## Chapter 3. XML 出力

Cppcheck はXML形式によるデータ出力も可能です。

Cppcheckを実行する際に `-xml` フラグを使用してください。

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

XMLフォーマットサンプル:

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

属性について:

|                 |  |
|-----------------|--|
| <i>file</i>     | ファイル名  |
| <i>line</i>     | 行数   |
| <i>id</i>       | エラー番号  |
| <i>severity</i> | エラーの種類 (error / possible error / style / possible style) |
| <i>msg</i>      | エラーメッセージ   |

## Chapter 4. 出力の整形

出力を整形することもできます。

その場合はテンプレートを使用します。

VisualStudioライクな出力形式にしたい場合は”`--template vs`” を使うとよいでしょう。

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

出力結果はこのように表示されます。

```
Checking gui/test.cpp...
```

```
gui/test.cpp(31): error: Memory leak: b
```

```
gui/test.cpp(16): error: Mismatching allocation and deallocation: k
```

gccライクな出力形式にしたい場合は”`--template gcc`” を使うとよいでしょう

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

出力結果はこのように表示されます。

```
Checking gui/test.cpp...
```

```
gui/test.cpp:31: error: Memory leak: b
```

```
gui/test.cpp:16: error: Mismatching allocation and deallocation: k
```

オリジナルフォーマットで出力することもできます。

(例えばcsv形式だと以下のようになります):

```
cppcheck --template "{file},{line},{severity},{id},{message}"
```

```
gui/test.cpp
```

出力結果はこのように表示されます。

```
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. エラー無視

特定のバグの種検出報告を無視することもできます。

無視するエラーを指定するファイルを作成する必要があります。

形式は以下の通りです。

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

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

```
[error id]
```

error id は無視したいエラーのIDです。--XMLコマンドを使ってIDを割り出しコピーしてください。

例

```
memleak:file1.cpp
```

```
exceptNew:file1.cpp
```

```
uninitvar
```

以下のように実行することでエラー無視設定を有効にできます。

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

## Chapter 6. メモリリーク

メモリリークとリソースリークを探すのは、Cppcheckの重要な特色です。

Cppcheckはデフォルトで多くの一般的なリークの気配を検出できます。しかし万能ではありません。

あなたのちょっとしたサポートがあれば、Cppcheckはさらに強力になり、誤ったエラー報告も減少させることができます。

### 自動解放するヤツら

誤ったエラー報告の多くは、自動で解放するクラスなのにメモリリークと扱ってしまうパターンです。

サンプルコード:

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

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

Cppcheckはこのコードから解放処理をみつけることができません。  
なので以下のような警告を出力します。

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

前述のように”possible”キーワードはエラー報告に確信が持てない場合に出力されます。

自動解放機能をもっているクラスはテキストファイルに書き出しておいてください。

予めCppcheckにその情報を教えてあげることで、自動解放かどうかを判断することができるようになります。

```
(qt.lst)
QLabel
QPushButton
```

実行時に `-auto-dealloc` オプションを与えてください。

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

## ユーザー定義のアロケート関数

Cppcheck は多くの一般的なメモリアロケーション関数を知っています。(malloc free new delete...etc) しかし、あらゆるアロケーション関数を把握しているわけではありません。.

メモリリークしそうなサンプルコード:

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

上記のコードからCppcheckはリークの気配を感じ取りません。

たとえ以下のようにpossibleフラグをつけてもCppcheckの鼻の良さは変わらないのです。

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

CreateFred や DestroyFred がメモリアロケート関数だと認識していないのです。

別ファイルで以下のように定義していたとしましょう。

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

我々人間には先程のサンプルコードが明らかにリークしている危険なコードだと分かります。

そしてCppcheckも、この関数の内容を見ることさえできれば、そのことを理解できます。

このようにCreateFred()が内部でアロケートしたメモリを返すことを、そしてDestroyFred()が正しく解放していることをCppcheckに“見せる”必要があります。

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

出力結果：

```
Checking fred1.cpp...
```

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

## Chapter 7. 例外安全

Cppcheckには、あなたのコードが例外安全であることを保証するいくつかのチェックがあります。

サンプルコード

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

デフォルトで、cppcheckはこのコードのどんな問題も検出しません。

例外安全なチェックを可能にするために、`--enable` を使います。

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

出力結果

```
[fred.cpp:3]: (style) Upon exception there is memory leak: a  
If an exception occurs when b is allocated, a will leak.
```

別の例を見てみましょう。

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

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

出力結果

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