Cppcheck で C/C++コードの静的解析を行う

TN0002-00／2023 年 10 月 28 日／こがねさん(著)
https://www.kumikomist.com/

■目次

■文書内の記号

	取り扱いにおける禁止事項（してはいけないこと）を示しています。
	取扱における指示事項（必ずしなければいけないこと）を示しています。

	取り扱いにおける注記事項を示しています。
	取り扱いにおけるポイントを示しています。

1. はじめに

Cppcheck は、C/C++用のフリーの静的解析ツールです。静的解析とはプログラムを実行せず、ソースコードそのものを解析して誤りや異常の発生しそうな箇所を特定するというものになります。

Cppcheck は商用のものと比べると機能は大きく劣りますし、フォントを変えれなかったり警告メッセージが英語であったり、ソースコードの全角文字が化けることもあるなど不便なところも多くあります。しかしほぼ無設定で簡単に解析が行えるため、テスト前には必ず通すようにしています。

2. 必要なもの

- Cppcheck
  http://cppcheck.sourceforge.net/

3. 設定

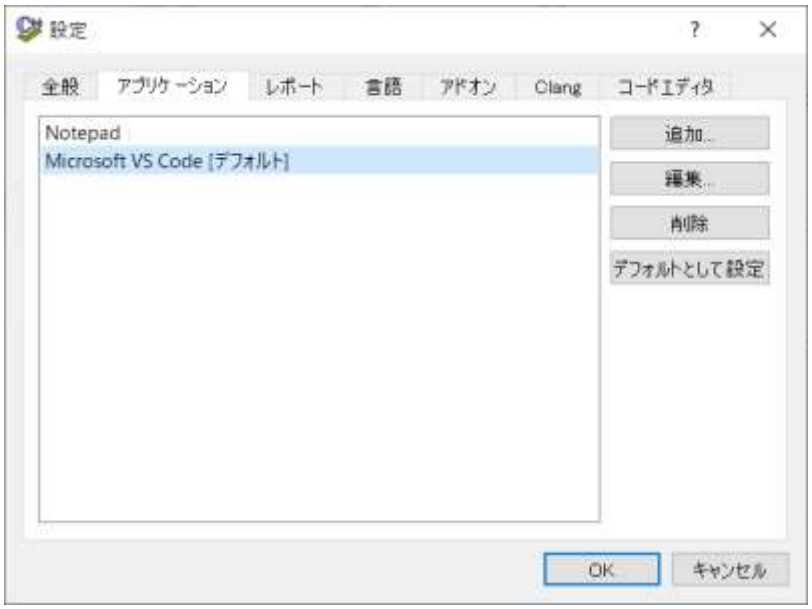
設定はいくつかありますが、本当に必要なものは 1 つだけです。使用するエディターを設定します。

デフォルト設定はメモ帳になっており、指摘箇所（行）へ飛ぶことができません。このためコマンドラインにより指定した行で開くことのできるエディターが必須となります。「Visual Studio Code」「Mery」「サクラエディタ」などを用意してください。

	● この設定は、Cppcheck をバージョンアップする毎に必要です。

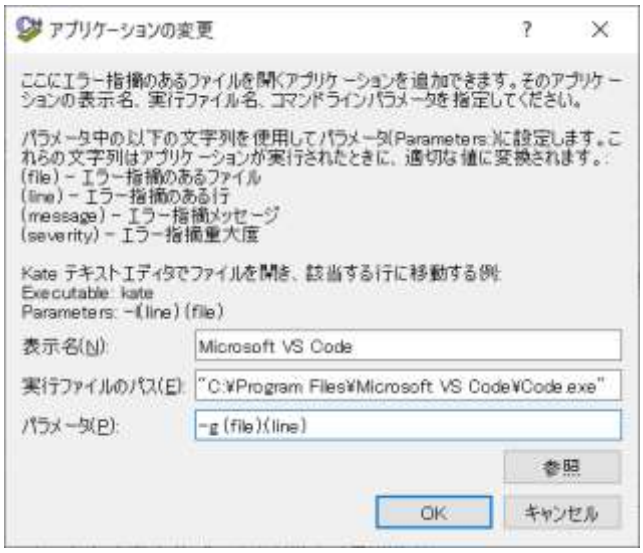
(1) メニュー [編集 > 設定] を開きます。

(2) アプリケーションタブを開きます。



(3) ここで使用するエディターを設定しますが、Visual Studio Code とそれ以外で操作が変わります。Visual Studio Code の場合は表示されている名前を選択して、[デフォルトとして設定] ボタンをクリックするだけで済みます。それ以外のエディターを使用する場合は、[追加] ボタンをクリックして次へ進んでください。

(4) 下記のウィンドウが表示されるので、[参照] ボタンをクリックしてエディターの exe へのパスを通します。



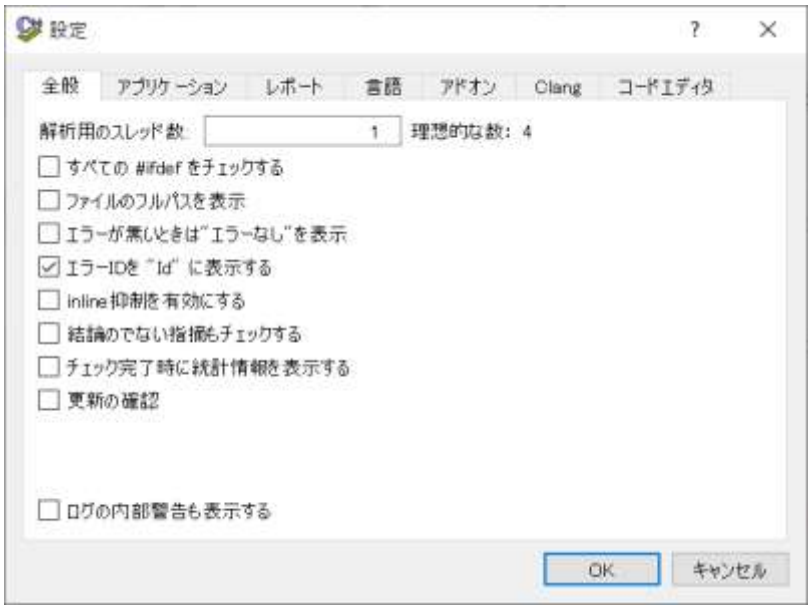
(5) 続いてパラメータにコマンドラインを入力します。こちらは使用するエディターによって入力方法が変わるので、頑張って調べてください。参考までに、いくつかエディターの例を示しておきます。

・EmEditor：(file) /l (line)
・Mery：/l (line) (file)
・サクラエディタ：(file) -Y=(line)

3.1. その他推奨設定

(1) メニュー [編集 > 設定] を開きます。

(2) [エラーID を "Id" に表示する] にチェックを付けます。



4. 使用方法

メニュー [チェック > ディレクトリ選択] にて、ソースコードを保存しているフォルダーを選択します。これだけで解析を開始し、異常を検知したファイルがあれば表示します。

⚠	● ソースファイルのエンコードは「UTF-8 with BOM」でなければ全角文字が化けます。「UTF-8」ではいけないようです。ただし化けるだけで解析には影響しないと思います。

解析の結果指摘事項が見つかると、図 1 のような表示となります。

ここで①の指摘をクリックすると、②に指摘の概要と、③に実際の該当箇所が表示されます。また①をダブルクリックすることで、設定で指定したエディターにて指摘箇所のソースファイルを開きます。

図 1

5. 指摘

以降に Cppcheck が検出する指摘の例を紹介します。
見出しとなっている「uninitvar」などの文字列は Cppcheck の示すエラーID です。設定でエラーID を表示するにチェックを付けると、表示されるようになります。
　また「CWE」は Common Weakness Enumeration の略で、共通脆弱性タイプです。ソフトウェアやハードウェアの脆弱性をパターン化し、番号を割り振って識別したものとなります。Cppcheck ではエラーID に該当するCWE-ID も表示します。

5.1. 警告

5.1.1. invalidPrintfArgType_sint（CWE-686）

フォーマットの型指定が間違っています。型を揃えてください。

/* 適合例 */
unsigned int num = 0;
sprintf(buf, "%u", num);
	/* 不適合例 */
	int num = 0;
	sprintf(buf, "%u", num);

5.1.2. oppositeInnerCondition（CWE-398）

デッドコードがあります。

```
/* 不適合例 */
if (a == 0) {
  if (a == 1) {
    // ここには来れない
  }
}
```

5.1.3. pointerSize（CWE-467）

ポインター変数に対して sizeof 演算子を使ってはいけません。

/* 適合例 */
void func(char str[], int size)
{
  memset(str, 0x00, size);
}
	/* 不適合例 */
	void func(char str[])
	{
	  memset(str, 0x00, sizeof(str));
	}

5.1.4. uninitvar（CWE-457）

変数を初期化していません。値を初期化してから使用してください。

5.2. スタイル

5.2.1. clarifyCondition（CWE-398）

( )で囲み条件を明確にしてください。

/* 適合例 */
if ((data & 0x01) != 0) {
}
	/* 不適合例 */
	if (data & 0x01 != 0) {
	}

5.2.2. constParameter（CWE-398）

関数内で書き換えないポインター変数は const 宣言すべきです。

/* 適合例 */
void func(const char buf[], int len) {
  // 関数内で buf の値は書き換えない
}
	/* 不適合例 */
	void func(char buf[], int len) {
	  // 関数内で buf の値は書き換えない
	}

5.2.3. redundantCondition（CWE-398）

条件が常に真です。

```
/* 不適合例 */
if ((x == 0 ) && (x != 1)) {
}
```

5.2.4. redundantAssignment（CWE-563）

変数を使用する前に上書きしています。

5.2.5. unreadVariable（CWE-563）

変数に格納した値を使用していません。

5.2.6. unusedFunction（CWE-561）

関数を使用していません。未使用の関数は削除してください。

5.2.7. unusedVariable（CWE-563）

変数を使用していません。未使用の変数は削除してください。

5.2.8. variableScope（CWE-398）

変数のスコープを縮小できます。

特定のブロック内でしか使用しない変数は、ブロック内で宣言することで保守性が向上します。

/* 適合例 */
while (1) {
  int ret;
  ret = function(x);
  if (ret == 0) {
  break;
  }
  ...
}
	/* 不適合例 */
	int ret;
	while (1) {
	  ret = function(x);
	  if (ret == 0) {
	  break;
	  }
	  ...
	}