|  |
| --- |
|  |
| Cppcheck-1.70分析报告 |
|  |
|  |
| **BITion小组：李铮、过其靖、虞竞、赵峻瑶、孙云霄** |
|  |
| **2016/4/8** |

目录

1. Cppcheck简介 2

2. Cppcheck流程分析 3

2.1. Cppcheck总体程序逻辑 3

2.2. Cppcheck文件分析流程 4

3. Cppcheck功能模块 6

3.1. Cppcheck代码文件读入 6

3.1.1. 命令行参数解析 6

3.1.2. 文件读取（ParseFromArgs函数） 8

3.1.3. 文件预处理模块（Preprocessor类） 10

3.2. Cppcheck词法分析/语法分析（SymbolDatabase类） 11

3.2.1. SymbolDatabase主要定义类型 11

3.2.2. Variable 12

3.2.3. Scope 16

3.2.4. Function 19

3.2.5. ValueType 19

3.2.6. Token 21

3.2.7. Value 28

3.3. 缺陷报告（ErrorLogger类） 31

3.4. 缺陷检测（Check类及其派生子类） 32

3.4.1. CppCheck缺陷库 32

4. Cppcheck-1.70版本缺陷分析 42

5. 参考文献 47

# Cppcheck简介

Cppcheck是针对C/C++语言的静态软件缺陷检测工具，它与其他的一些C/C++编译器或静态分析工具不同，它不检查语法错误，其主要检测那些编译器不能检查出来的一些高级代码缺陷。Cppcheck采用内嵌检查类实现了缺陷模式检测，然后根据词法分析阶段Token对象中变量linenr的值来精确定位出缺陷。Cppcheck将软件缺陷从静态源代码的角度划分若干个检查类，分别给予检测和定位，并赋予8种不同的级别，分别为: none，error，warning，style，performance，portability information，debug。Cppcheck的实现主要包含如下几个方面:

(1)对源代码进行适当的处理和化简，便于后面的缺陷模式检测；

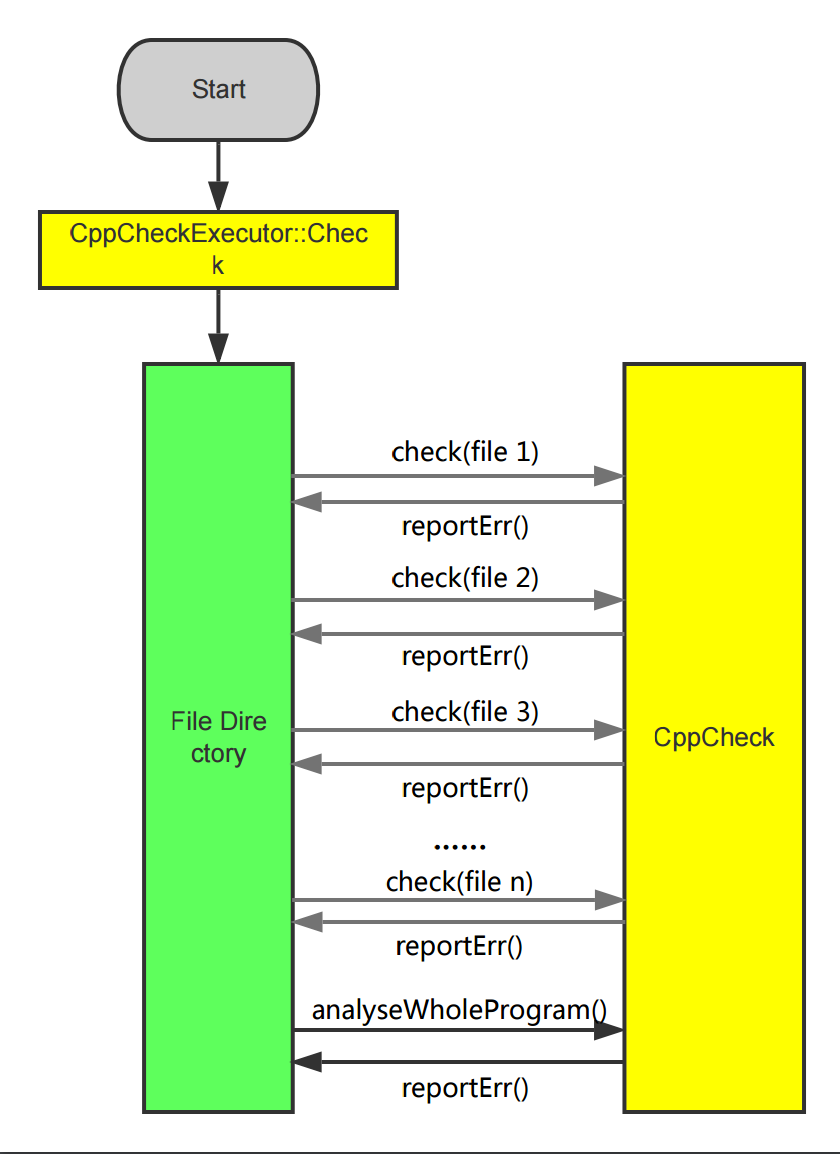
(2)定义一种缺陷，并将其模式提取出来，并用合理的正则表达式表示出来；

(3)对提取出来的缺陷模式，采用C++编程技术，在源代码中搜索，检测和定位。

静态缺陷检测和定位的关键技术就在于词法分析阶段的词法分析以及语义分析环节的优良设计。由于Cppcheck采用了开源正则表达式库PCRE，使得词法分析阶段的缺陷模式的匹配和定位功能强大。Cppcheck采用PCRE实现了词法分析阶段缺陷的检测，然后根据Token对象中变量linenr的值实现了缺陷的定位。由于缺陷模式的可扩充性，也导致了Cppcheck良好的扩展性。PCRE正则表达式的模式匹配的特点，使得Cppcheck检查出来的缺陷都是模式相同的一类缺陷，而不是单一的特定缺陷。此外Cppcheck 还提供了扩充接口，可在原有基础上对其进行扩充，进一步提高缺陷检测能力。

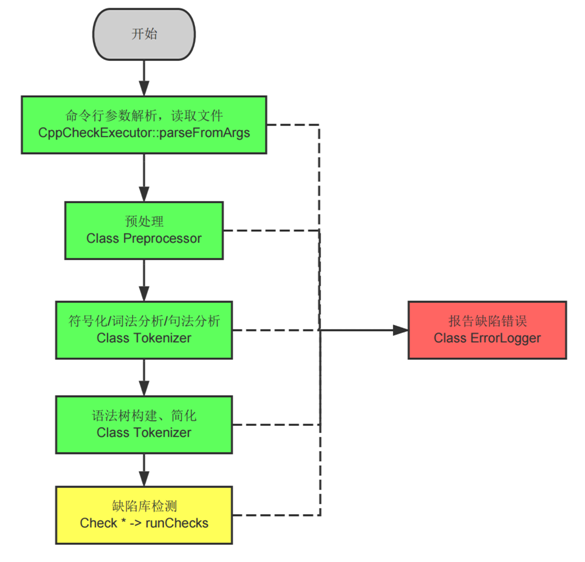
# Cppcheck流程分析

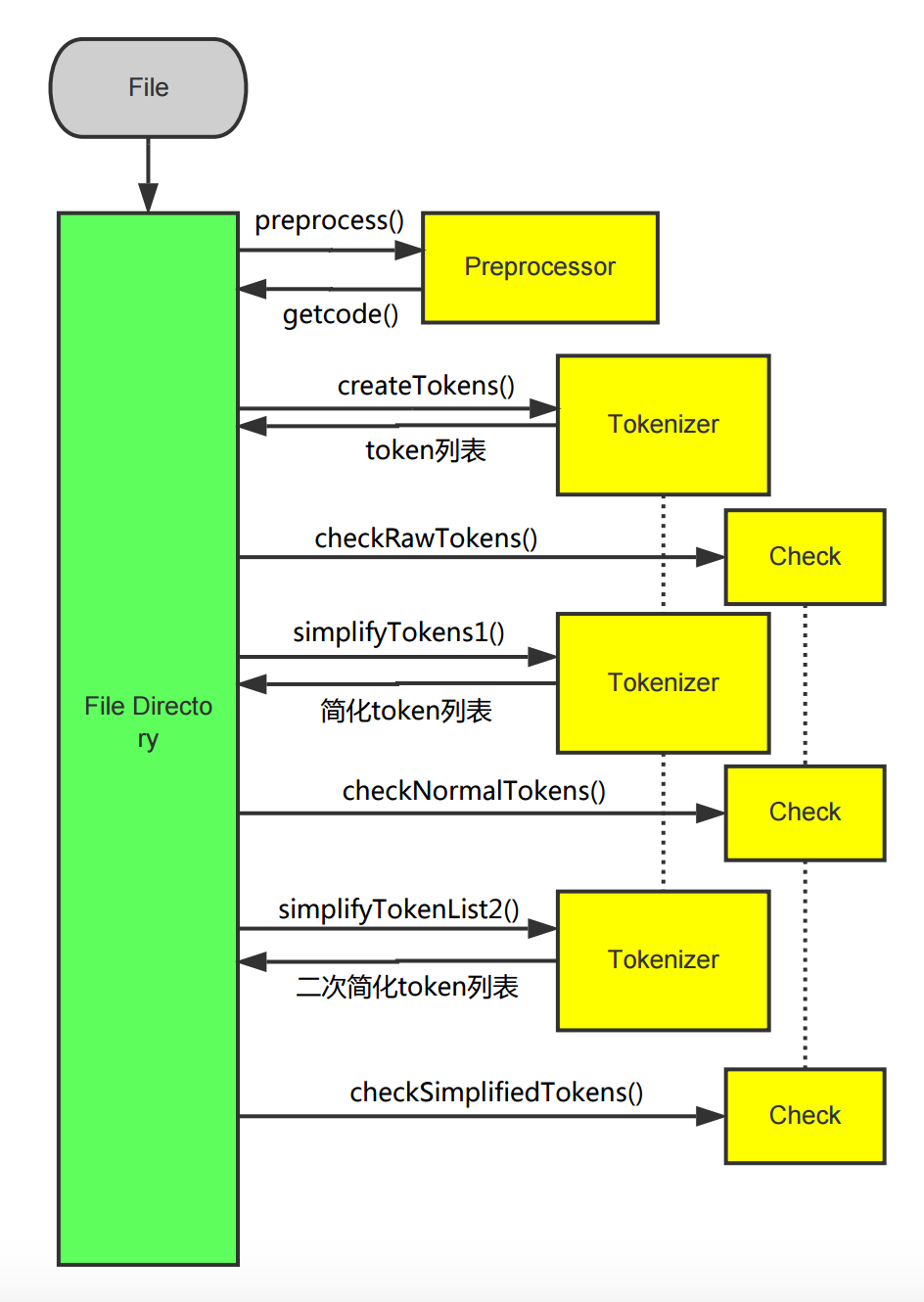
## Cppcheck总体程序逻辑

****

CppCheck主程序通过创建一个CppCheckExecutor并执行其中的Check方法，对工程目录内的每个文件调用检查，每检查一个文件回调reportErr方法输出发现的错误，当完成所有文件的分析之后，调用analyseWholeProgram方法分析整个工程，检查一些全局缺陷（全局未使用函数，变量等）。

## Cppcheck文件分析流程





CppCheck对于单个文件的处理主要是先调用 Preprocessor 的 preprocess 方法进行预处理，然后调用 getcode方法得到预处理后的代码，并使用Tokenizer 的createToken方法进行词法分析，并将初步分析和化简完的代码交给各Checker进行第一部分的分析checkRawToken; 接下来通过Tokenizer的simplifyTokens1对代码进行简单的包括语法树删减，常量变量替换等化简，之后再进行第二部分的分析checkNormalTokens。最后调用simplifyTokenList2方法对token列表进行进一步简化， 该部分简化按照部分缺陷库逻辑将code中已认为不存在错误可能的词元全部删除，并进一步简化运算，指针操作等，最后checkSimplifiedTokens再对简化后的token列表进行检查。

# Cppcheck功能模块

## Cppcheck代码文件读入

### 命令行参数解析

以该附test目录下测试文件为例执行cppcheck

*cppcheck.exe --enable=all ~\cppcheck\test\cfg\gnu.c*

1. cppcheck.exe为可执行文件名，默认为命令行输入第0个参数（即argv[0]）
2. --enable=all为设置，可根据用户需要提供多种设置，比较重要的设置有（详见项目下-help输出）

|  |  |
| --- | --- |
| 设置字段 | 说明 |
| —enable=<id> | 允许额外检查 |
| <id> all | 允许所有检查（cppcheck推荐）扫描整个解决方案 |
| <id> warning | 允许输出警告消息 |
| <id> style | （与performance、portability相关） |
| <id> information |  |
| <id> unusedFunction | 允许检查未使用函数（cppcheck推荐在检查整个工程时开启） |
| <id> missingInclude | 允许检查未include的文件 |
| —library=<cfg> | 从.cfg（xml）文件中读取更多有关工程文件类型与函数的信息，更好地了解代码，默认加载std.cfg |
| -i <dir or file | 不检查给定文件或给定目录下的所有文件 |
| -q, —quiet | 分析过程中不显示进度 |
| -rp=<paths>, --relative-paths=<paths> |  |
| —suppress=<spec> | 抑制与<spec>格式匹配的错误报告 |
| <spec> | [error id]:[filename]:[line |
| —suppressions-list=<file | 抑制list中文件的错误报告 |
| —template=<text> | 按照text格式输出错误报告  e.g.  '{file}:{line},{severity},{id},{message}' or  '{file}({line}):({severity}) {message}' or  '{callstack} {message}’  已预定义template模板：gcc，vs，edit |
| —xml | 将错误报告写入xml文件 |

1. ~\cppcheck\test\cfg\gnu.c为待检测目录，可为单个.cpp/.c文件或整个工程目录

### 文件读取（ParseFromArgs函数）

主要功能：解析命令行参数，设置和文件列表，输入错误时返回错误。

实现：

|  |
| --- |
| * *Settings& settings = cppcheck->settings()* |

一个通用设置的容器，这样我们就不需要将单个值传递给函数或构造函数，或者在将来可能有更详细的设置。

|  |
| --- |
| * *CmdLineParser parser(&settings)* |

命令行语法分析器：分析用户在命令行提供的选项和参数。该解析器使用一个指针，它将根据用户的选择来更新它的设置实例。设置将根据选项修改用户的设置。

|  |
| --- |
| * *const bool success = parser.ParseFromArgs(argc, argv)* |

分析给定的命令行：如果命令行是正确的，返回true;反之，返回false。

|  |
| --- |
| * *(parser.GetShowVersion() && !parser.GetShowErrorMessages())* |

判断用户是否希望看到程序版本以及用户是否想要查看错误消息列表

|  |
| --- |
| * *const char \* extraVersion = cppcheck->extraVersion()* |

额外的描述信息，如Git提交ID，建造时间/日期等。

|  |
| --- |
| * *const std::string path(Path::toNativeSeparators(\*iter))*   *if (!FileLister::isDirectory(path))*  *{*  *iter = settings.\_includePaths.erase(iter);*  *}* |

检查所有包含路径是否存在

|  |
| --- |
| * *const std::vector<std::string>& ignored = parser.GetIgnoredPaths()*   *if (Path::isHeader(\*i))*  *{*  *warn = true;*  *}* |

如果用户试图排除头文件，输出一个警告

|  |
| --- |
| * *FileLister::recursiveAddFiles( \_files,Path::toNativeSeparators(\*iter),*   *\_settings->library.markupExtensions(), matcher)* |

给定recursiveaddfiles()每个给定的文件参数,实现递归添加文件

### 文件预处理模块（Preprocessor类）

| **主要功能** | **实现函数** |
| --- | --- |
| 读取代码文件 | *string getcode(conststd::string&filedata, conststd::string&cfg, conststd::string&filename)* |
| 去除多余空格 | *void preprocessWhitespaces(std::string&processedFile)* |
| 删除汇编代码 | *void removeAsm(std::string&str)* |
| 替换宏定义 | *string expandMacros(conststd::string&code, std::string filename, conststd::string&cfg, ErrorLogger \*errorLogger)* |
| 提取预处理配置设置 | *bool validateCfg(conststd::string&code, conststd::string&cfg)* |
| 处理include内容及嵌套 | *string handleIncludes(conststd::string&code, conststd::string&filePath, conststd::list<std::string>&includePaths, std::map<std::string,std::string>&defs, std::set<std::string>&pragmaOnce, std::list<std::string> includes)* |

Preprocessor类用read方法从上一模块中解析出的代码文件中读入代码字符串，依照用户输入配置设置，preprocess方法对代码通过字符串解析处理实现上述功能，然后调用getcode获得经过预处理之后的代码，供之后的模块使用。头文件处理方面，Preprocessor类中设置了关于头文件的枚举类HeaderType，并在missingIclude方法中判断代码文件中引用的文件是否存在。同时，类中许多方法中通过调用一个ErrorLogger类指针将头文件不存在，代码宏定义出错等错误记录，并通过wirteError方法输出。

## Cppcheck词法分析/语法分析（SymbolDatabase类）

### SymbolDatabase主要定义类型

为组织读入的代码文件，CppCheck在SymbolDatabase类中定义了一些词元类，这些类之间的基本彼此都存在使用，继承或友元关系，较为复杂。

|  |  |
| --- | --- |
| 类名 | 描述 |
| ValueType | 变量类型 |
| Token | 词元 |
| Scope | 语句域 |
| Variable | 变量信息 |
| Type | 用户定义Class类信息 |
| Function | 函数信息 |

SymbolDatabase同时还拥有这些类型的列表，容器及索引get函数用来在相应数据结构中获取相应某一词元对象。

|  |  |
| --- | --- |
| 变量名 | 类型 |
| scopeList | list <Scope> |
| functionScopes | vector <const Scope \*> |
| classAndStructScopes | vector <const Scope \*> |
| typeList | list <Type> |
| \_variableList | list <Type> |

### Variable

Variable是一个存储变量信息的类，主要有以下成员变量：

|  |  |  |  |  |  |  |
| --- | --- | --- | --- | --- | --- | --- |
| 成员变量名称 | 类型 | 含义 | | | | |
| \_name | Token\* | 变量名词元指针（该变量第一次被声明的token） | | | | |
| \_start | Token\* | 变量定义起始词元指针 | | | | |
| \_end | Token\* | 变量定义结束词元指针 | | | | |
| \_index | Size\_t | 变量在所在scope中的索引（从0开始自增） | | | | |
| \_access | <enum> AccessControl | 可访问类型，共有8种： | | | | |
| public | protected | private | | global |
| namespace | argument | local | | throw |
| \_flag | unsigned int | 信息标志位，已使用的共有14位： | | | | |
| 类型 | 说明 | | | |
| fIsMutable | 有mutable关键字 | | | |
| fIsStatic | 有static关键字 | | | |
| fIsConst | 有const关键字 | | | |
| fIsExtern | 有extern关键字 | | | |
| fIsClass | 是class型变量 | | | |
| fIsArray | 是数组变量 | | | |
| fIsPointer | 是指针变量 | | | |
| fIsReference | 是否是引用型（&）变量 | | | |
| fIsRValueRef | 是否有右值引用（c++ 11）关键字 | | | |
| fHasDefault | 是否有默认值（缺省参数） | | | |
| fIsStlType | 是否为标准库变量 | | | |
| fIsStlString | 是否为标准库字符串 | | | |
| fIsIntType | 是否为整形 | | | |
| fIsFloatType | 是否为浮点数类型（包括float，double） | | | |
| \_type | Type\* | 如为用户定义类，详见Type类介绍 | | | | |
| \_scope | Scope\* | 变量所在域的指针 | | | | |
| \_dimensions | vector<Dimension> | “维度”类容器，以存储数组信息，  附：Dimension类成员变量说明 | | | | |
| 成员变量名 | 类型 | | 说明 | |
| start | Token\* | | 多维数组起始词元 | |
| end | Token\* | | 多维数组结束词元 | |
| num | MathLib::bigint | | 维度的大小，0表示未知 | |
| known | bool | | 维度大小已知 | |

附：若干Variable对象例

测试代码（以下代码中的“:”均应为“->str=”，为说明简便因此简写）

|  |
| --- |
| class myclass  {  float c;  // start:float end:float index:0 flags:fIsFloatType  // access:Private    public:  double d;  // start:double end:double index:1 flags: fIsFloatType  // access:Public  };  struct mystruct  {  int e;  // start:int end:int index:0 flags:fIsIntType  // access:Public    protect:  double f;  // start:double end:double index:1 flags:fIsFloatType  // access:Protected  };  mystruct mstruct;  // start:mystruct end:mystruct index:0 flags:fIsClass  // access:Global type:非空  extern int\* q;  // start:int end:\* index:1  // flags:fIsIntType,fIsPointer,fIsExtern access:Global  int main(void)  {  int ix = 0;  // start:int end:int index:0 flags:fIsIntType  // access:Local  const float iy = 1;  // start:float end:float index:1  // flags:fIsFloatType,fIsConst access:Local  static double iz = 2;  // start:double end:double index:2  // flags:fIsFloatType,fIsConst access:Local    std::string str[100][200][300];  // start:std end:string index:3  // flags:fIsStlString,fIsStlType,fIsArray,fIsClass  // access:Local  // dimensions[0].num = 100,  // dimensions[1].num = 200,  // dimensions[2].num = 300  } |

### Scope

Scope表示域的概念，可以简单地理解为一对“{}”之间为一个域，Scope内置一个枚举型变量ScopeType，用于表示Scope的类型。

|  |  |
| --- | --- |
| 枚举名 | 说明 |
| eGlobal | 全局域，默认类型 |
| eClass | class域（class{}） |
| eStruct | struct域（struct{}） |
| eUnion, | 枚举域（union{}） |
| eNamespace, | 命名空间（namespace{}） |
| eFunction | 函数体域 |
| eIf | if域 |
| eElse | else域（else域之后会自动添加一个try域） |
| eFor | for域 |
| eWhile | while域 |
| eDo | do域 |
| eSwitch | switch域 |
| eUnconditional | 无明确含义域（单纯用”{}”包括） |
| eTry | try域 |
| eCatch | catch域 |
| eLambda | lambda简单替换函数域 |

Scope内置的主要成员变量：

|  |  |  |
| --- | --- | --- |
| 成员变量名 | 类型 | 说明 |
| check | SymbolDatabase\* | 指向全局symmboldatabase |
| className | string | 函数域或class/struct域的名称 |
| classDef | Token\* | 域的定义起始token |
| classStart | Token\* | 域起始token（通常为”{”） |
| classEnd | Token\* | 域结束token（通常为”}”） |
| functionList | list<Function> | 域中函数列表 |
| functionMap | multimap<string, Function\*> | 域中函数名及Function对象红黑树对 |
| varlist | list<Variable> | 变量列表（使用较少） |
| nestedIn | Scope\* | 域的上一级域指针 |
| nestList | list<Scope \*> | 从属域列表（使用较少） |
| numConstructors | int | 域中构造函数个数 |
| numCopyorMoveConstructors | int | 域中拷贝/赋值构造函数 |
| type | ScopeType | 域类型 |

附：简要域说明代码

|  |
| --- |
| namespace test { //scope 1 type : eNamespace    class A //scope 2 type : eClass  {  int x;  int y;    A()  { //scope 3 type : eFunction  x = 1;  y = 2;  } //end scope 3  }; //end scope 2      const int fun2() //scope 4 type : eFunction  {  int a;  return a;  } //end scope 4  float fun3() //scope 5 type : eFunction  {  return 1.11;  } //end scope 5    int main(void) //scope 6 type : eFunction  {  int i = 1;  if(i) //scope 7 type : eIf  {    } //end scope 7    else if(i) //scope 8 type : eElse  //scope 9 type : eTry  //scope 10 type : eIf  {    } //end scope 8,9,10  else //scope 11 type : eElse  //scope 12 type : eTry  {    } //end scope 11,12      return 0;  } //end scope 6    } //end scope 1 |

### Function

### ValueType

ValueType 类主要用来存储token的类型及指针信息，其中的成员变量和方法均为public类型，ValueType虽然在SybolDatabase中被声明，但并没有作为成员变量在其中被调用，而是作为成员变量出现在Token类中。只有当token为type枚举类中存在的类型时其valuetype指针才不为空，其他token（如”int”，”;”或其他type枚举类中不存在的类型token）的valuetype指针均为空。

|  |  |  |  |
| --- | --- | --- | --- |
| 成员变量名 | 类型 | 说明 | |
| sign | <enum> Sign | token（Variable）的符号状态 | |
| 取值 | 说明 |
| UNKNOWN\_SIGN | 默认状态 |
| SIGNED | 有符号  （显式声明signed变量，字符变量，char字符串，int等声明均会被赋为此标记） |
| UNSIGNED | 无符号  （特指明确显式声明的unsigned变量） |
| type | <enum> Type | token（Variable）的类型 | |
| 取值 | 说明 |
| UNKNOWN\_TYPE | 默认类型 |
| NONSTD | C语言struct |
| BOOL | bool |
| VOID | void  （特指void \*） |
| CHAR | char |
| SHORT | short int |
| INT | int |
| LONG | long int |
| LONGLONG | long long int |
| FLOAT | float |
| DOUBLE | double |
| LONGDOUBLE | long double |
| pointer | unsigned int | 指针级别，0表示非指针 | |
| constness | unsigned int | 位图，每位对应表示每级指针指向是否为const，第0位表示原始数据。 | |
| （如const int\* const x = &a） x的ValueType->constness = 3（00000011），表示x指向位置不可修改，且x指向位置的数值不可修改 | |
| typeScope | Scope\* | 定义域，通常为空域 | |
| originalTypeName | string | 类型曾用名，通常为空串 | |

### Token

Token为CppCheck内语法结构的基本单位，代码中的每条语句均被拆解为若干token以双向链表，二叉树等若干数据结构保存。Token类中存有一个ValueType指针，ValueType类分析中已提及。Token中属性及标志位介绍如下：

枚举型tokType，用于标识token类型：

|  |  |  |
| --- | --- | --- |
| 类型 | | 说明 |
| 名称词元 | eVariable | 变量类型词元 |
| eType | 用户声明类型词元 |
| eFunction | 函数类型词元 |
| eKeyword | — |
| eName | 默认类型 |
| 终结词元 | eNumber | 数值常量类型 |
| eString | 字符串常量类型 |
| eChar | 字符常量类型 |
| eBoolean | 布尔常量类型 |
| eLiteral | c++11“字面值”关键字 |
| 运算符 | eArithmeticalOp | 包括移位运算和一般数值运算符 |
| eComparisonOp | 数值比较运算符 |
| eAssignmentOp | 赋值运算符 |
| eLogicalOp | 逻辑与或非 |
| eBitOp | 位运算与或非 |
| eIncDecOp | 自增自减 |
| eExtendedOp | 小括号，方括号等 |
| eBracket | 大括号以及尖括号（仅当\_link不为空时认为’<’或’>’为括号，否则认为是比较运算符） |
| 其它 | eOther | 不属于表中任何类型的token（包含’;’） |
| eNone | 空字符串类型 |

unsigned int \_flags标志位：

|  |  |
| --- | --- |
| 类型 | 说明 |
| fIsUnsigned | 显式声明为unsigned |
| fIsSigned | 显式声明为signed |
| fIsPointerCompare | 未发现引用 |
| fIsLong | 显式声明为long long或long long int |
| fIsStandardType | 所有std标准类型 |
| fIsExpandedMacro | 被宏定义替换时加上该标记 |
| fIsCast | — |
| fIsAttributeConstructor | GCC attribute 优化对齐 |
| fIsAttributeDestructor |
| fIsAttributeUnused |
| fIsAttributePure |
| fIsAttributeConst |
| fIsAttributeNoreturn |
| fIsAttributeNothrow |
| fIsAttributeUsed |
| fIsOperatorKeyword | 运算符重载 |
| fIsComplex | C++ std::Complex复数类型 |

Token类提供了若干GetFlags，SetFlags，和IsXX方法来设置，获取，判断某一标志位。

Token所在文件情况：

|  |  |  |
| --- | --- | --- |
| 成员变量名 | 类型 | 说明 |
| \_progressValue | unsigned int | 当前token在原文件中粗略位置百分比 |
| \_fileIndex | unsigned int | token所在文件索引 |
| \_linenr | unsigned int | token在当前文件中行号（预处理前） |

varId标识：

varId属性仅当token的tokType为eVariable时不为0，用于标识变量。对于其他类型的token，其varId为0。token的tokType被修改时，其varId也同时被更新。

union {

const Function \*\_function;

const Variable \*\_variable;

const ::Type\* \_type;

}

token类内置一个无名联合，当token的tokType为eFunction，eVariable或结构体时，可通过直接访问该联合内的指针查询对应类型对象。

std::list<ValueFlow::Value> values：

当Token为变量时,values列表存储着该变量所有可能的取值，详见ValueFlow类分析。同时Token中提供了若干方法从values列表中检索可能取值的函数。

|  |  |
| --- | --- |
| 成员函数名 | 说明 |
| getValue | 查找传入值的Value对象，无则返回空指针 |
| getMaxValue | 查找该变量token可能拥有的最大值的value对象 |
| getValueLE | 查找token中值小于等于传入值的value对象 |
| getValueGE | 查找token中值大于等于传入值的value对象 |
| getValueTokenMaxStrLength | 当token为string型变量时可从其value列表中获取到其所有可能取值中长度最大的token |
| getValueTokenMinStrSize | 当token为string型变量时可从其value列表中获取到其所有可能取值中长度最小的token |
| getValueTokenDeadPointer | 当该token为指针时，可判断该指针是否为Dead Pointer[[1]](#footnote-1)，如是则返回其指向变量的起始token |

Token组织形式：

Token以双向链表形式组织，同时用scope存放该Token的所在域。

|  |  |  |
| --- | --- | --- |
| 成员变量名 | 类型 | 说明 |
| \_next | Token\* | 前驱Token |
| \_previous | Token\* | 后继Token |
| \_link | Token\* | 另一半括号 |

Token提供了若干用于在访问Token的方法：

|  |  |
| --- | --- |
| 函数变量名 | 说明 |
| tokAt | 返回距当前token传入int型位置的token，当前位置为0，前驱位置索引为负值，后继为正值 |
| linkAt | 返回tokAt->link |

Token中还提供了用于建立其语法树的若干指针，方法：

|  |  |  |
| --- | --- | --- |
| 成员变量名 | 类型 | 说明 |
| \_astOperand1 | Token\* | 语法树左节点 |
| \_astOperand2 | Token\* | 语法树右节点 |
| \_astParent | Token\* | 语法树父节点 |

|  |  |
| --- | --- |
| 函数变量名 | 说明 |
| \_astOperand1 | 返回语法树左节点 |
| \_astOperand2 | 返回语法树右节点 |
| \_astParent | 返回语法树父节点 |
| astTop | 返回语法树根节点 |
| isCalculation | 判断当前节点是否是一个计算节点 |
| argumentType | 回溯根节点，返回其返回值类型的ValueType指针 |

CppCheck中所使用的CppCheck抽象语法树（CppCheck Abstract Syntax Tree 下文简称CppAST）并非严格意义上的语法树（Syntax Tree）或抽象语法树（Abstract Syntax Tree），而是完全的二叉树。同时，CppAST中没有任何对if，else，switch，case包括for等控制流关键字的体现（其tokType与’;’相同均为eName），这也是CppCheck几乎没有对任何控制流代码内容进行check的原因，参见以下示例代码：

|  |
| --- |
| while(b != 0)  {  if (a > b)  a = a - b;  else  b = b - a;  }  return a; |

抽象语法树（AST）



CppAST：

而CppCheck对于上述代码的组织方式为将其按照逻辑行（CppCheck在Preprocessor过程中将自然行整理为逻辑行）拆解为5部分，构建了5棵CppAST，之后的简化，检查操作对其逐一处理。以下列出几种常见特色CppAST示例。

CppAST示例：

|  |  |
| --- | --- |
| 测试代码及说明 | CppAST示图 |
| int astFunc(int a，int b，int c) |  |
| CppAST对函数逻辑行的建立为统一将左括号’(’设为根节点（树中无’）’，但’(’的\_link不为空），同时将其返回值赋给根节点ValueType下的Type，其参数列表中所有的形参均会被注册入SymbolDatabase中。如图，通过中序遍历即可还原该函数声明。 |
| for (i = 0; i <= n; i++) |  |
| 对for循环体的处理与函数类似，但根节点’(’的ValueType为空。同时，对于自增自减等单条目运算，无论其与操作数的位置如何（i++与++i），操作数始终在左子树。 |

Token匹配机制：

Token中提供了若干种用于模式匹配的Match成员函数，同时提供多种匹配模式。

|  |  |
| --- | --- |
| 成员函数名 | 说明 |
| simpleMatch | 简单匹配，即pattern可匹配传入token及其之后连续token的str则成功 |
| Match | 完整的正则匹配，提供多种模式 |
| findsimplematch | 查找从startTok开始第一个能与pattern简单匹配的token |
| findmatch | 查找从startTok开始第一个能与pattern匹配的token |

|  |  |
| --- | --- |
| 匹配式 | 可匹配类型 |
| %any% | 任何token |
| %assign% | 赋值操作符 |
| %bool% | 布尔值（true/false） |
| %char% | 单引号包括的常量 |
| %type% | 变量类型 |
| %comp% | 比较运算符 |
| %cop% | 一般运算符，位运算符，逻辑运算符，比较运算符 |
| %name% | 变量或类型标识符 |
| %num% | 任何类型常数 |
| %or% | | |
| %op% | 任何运算符 |
| %oror% | || |
| %str% | 任何字符串常量 |
| %var% | 任何类型变量 |
| %varid% | 与参数varId相匹配的变量 |
| [abc] | 中括号中任意一个单字符 |
| int|void|char| | ’|’分割类型中任意一种，或不匹配任何类型 |
| int|void|char | ’|’分割类型中任意一种 |
| !!else | 不匹配除“!!”后类型的任何类型 |
| someRandomText | 任意与字符串参数相匹配的token |

以上各匹配式可通过’|’连接共同使用（推荐将%XX%类型前置）。

### Value

Value对象用于存储CppCheck推断的值，每个Token对象中存有一个Value列表，表示该token（变量或表达式）在当前CppAST下的的所有可能值。

|  |  |  |
| --- | --- | --- |
| 成员变量 | 类型 | 说明 |
| intvalue | long long | 用于存储整型值 |
| tokvalue | \*Token | 其他类型值，则直接存储其token的指针 |
| varvalue | long long | 节点计算值 |
| varId | unsigned int | 常量为0，当varvalue的取值为某变量在不同位置可能取值时会被赋相同varId，代表为同一变量 |

Value类内设置了valuekind枚举型变量，表示对该值的已知程度

|  |  |
| --- | --- |
| ValueKind枚举列表 | 说明 |
| Possible | 此value为一可能的取值 |
| Known | 此value为确定的取值 |
| Max | 此value为该变量在当前token中可能的最大值 |
| Min | 此value为该变量在当前token中可能的最小值 |

同时Value类还有conditional，inconclusive两个bool变量，不过当前版本下使用不完全，较少为true。

|  |  |  |
| --- | --- | --- |
| 成员变量 | 类型 | 说明 |
| conditional | bool | 推测为该value为满足一定条件下的取值，通常valuekind为Possible时为true |
| inconclusive | bool | 该value是否满足setting |

以下列出几种代码下变量token的values列表情况：

|  |  |
| --- | --- |
| int array[100];  int i = 8;  for (i = 0; i <= 100; i++)  {  array[i] = 0;      }  scanf("%d",&i);  scanf("%d",&i);    i = i > 0 ? 999:-999; | //token:array values为空  //token:i values为空  //三处 token:i values均为空  //token:array values为空  //token:i values[0]: intvalue=0 valuekind=Possible  // values[1]: intvalue=100 valuekind=Possible  //token:\"%d\" values[0]:intvalue=0 valuekind=Known  //token:i values[0]:intvalue=101 valuekind=Possible  //token:\"%d\" values[0]:intvalue=0 valuekind=Known  //token:i values为空  //第一处 token:i values为空  //第二处 token:i values[0]: intvalue=999  valuekind=Possible conditional=true  // values[1]: intvalue=-999  valuekind=Possible conditional=true |

可以看出，对于变量，只有当其被作为索引使用，取地址或作为比较运算操作数等情况时，当前token的values列表才会被创建。此外，当变量被输入时（scanf），其保存的value会被清空。

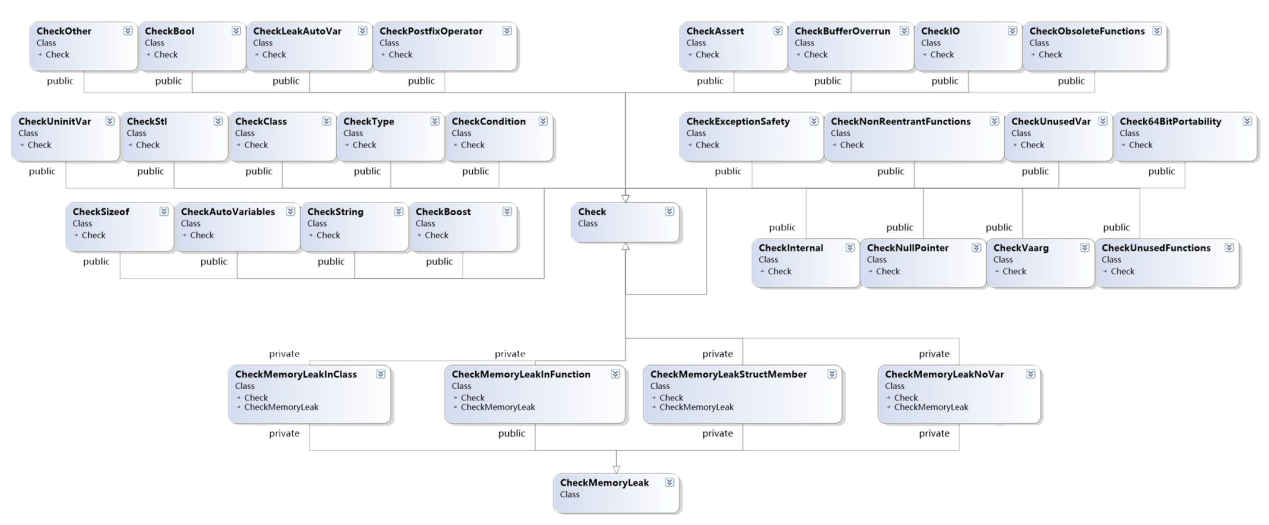
同时，valueFlow名空间中还提供了一些set，get，add，clear方法用于操作values。value的创建，add，反转，set方法具体实现细节非常复杂，此处不作具体解析。

## 缺陷报告（ErrorLogger类）

ErrorLogger主要用于统计，分类CppCheck类统计到的错误转化为消息输出。

## 缺陷检测（Check类及其派生子类）

### CppCheck缺陷库



**checknullpointer 检查空指针**

检查引用的指针、数组、成员变量/数组元素的地址、读写变量、字符串、流、循环、函数内是否有空指针，有空指针返回true，若未知是否引用指针则返回false。

**checkpostfixoperator 检查后缀运算**

cppcheck建议在非基本类型操作中使用++和—运算符作为前缀，而作为后缀通常用来保持前一个值的副本并添加一些额外的代码。

**checksizeof 检查sizeof**

|  |  |
| --- | --- |
| void sizeofsizeof() | 检查可疑代码sizeof(sizeof)，建议只使用一个sizeof，即类似于sizeof(size\_t) |
| void sizeofCalculation() | sizeof()中的计算 |
| void suspiciousSizeofCalculation() | 类似于sizeof()\*sizeof()或sizeof(ptr)/value的代码 |
| void checkSizeofForArrayParameter() | 作为函数参数的数组 |
| void checkSizeofForPointerSize() | 分配变量时的sizeof的使用 |
| void checkSizeofForNumericParameter() | 作为函数参数的数字 |
| void sizeofVoid() | sizeof(void)的使用 |

**checkstring 检查字符串**

|  |  |
| --- | --- |
| void stringLiteralWrite() | 未定义的行为和字符串 |
| void strPlusChar() | 错误的运算，将字符串与字符型的值相加 |
| void checkIncorrectStringCompare() | 错误的字符串比较，与strlen()结果不匹配 |
| void checkSuspiciousStringCompare() | 涉及char\*的比较，会警告是否应该使用strcmp()代替或引用指针 |
| void checkAlwaysTrueOrFalseStringCompare() | 不必要或可疑的字符串比较，结果总是true或false |
| void sprintfOverlappingData() | 传递给sprintf()的重叠的源和目标 |

**checktype**

|  |  |
| --- | --- |
| void checkTooBigBitwiseShift() | 右操作数过大的移位操作 |
| void checkIntegerOverflow() | 整数溢出 |
| void checkSignConversion() | 操作数可为负时的符号转换 |
| void checkLongCast() | int型的结果被分配或返回为long型 |

**checkother 其他**

|  |  |
| --- | --- |
| void clarifyCalculation() | 可疑的计算；不明确计算的优先级 |
| void clarifyStatement() | 可疑的声明 |
| void warningOldStylePointerCast() | 是否有C风格的指针在C++文件中 |
| void invalidPointerCast() | 不兼容的二进制表示 |
| void invalidFunctionUsage() | 无效的函数用法（无效的输入/重叠的数据） |
| void checkVariableScope();  static bool checkInnerScope(const Token \*tok, const Variable\* var, bool& used) | 检查变量范围 |
| void checkCommaSeparatedReturn() | 检查返回语句的逗号 |
| void checkConstantFunctionParameter() | 检查常函数参数 |
| void checkCharVariable() | 检查使用字符变量作为数组下标/位操作的操作数 |
| void checkIncompleteStatement() | 不完整的声明 |
| void checkZeroDivision() | 检查零部 |
| void checkNanInArithmeticExpression() | 检查算术表达式中的非数字 |
| void checkMathFunctions() | 检查数学函数中没有意义的参数 |
| void checkRedundantAssignment() | 冗余分配 |
| void checkRedundantAssignmentInSwitch() | Switch语句中的冗余分配 |
| void checkSuspiciousCaseInSwitch() | 可疑的case如’case A||B:’ |
| void checkSuspiciousEqualityComparison() | 可疑的等式 |
| void checkSwitchCaseFallThrough() | 失败的swich case |
| void checkMisusedScopedObject() | 检查被立即销毁的对象 |
| void checkMemsetZeroBytes() | memset()调用0字节的tmp |
| void checkMemsetInvalid2ndParam() | memset()中无效的第二参数 |
| void checkDuplicateBranch() | if和else分支相同 |
| void checkDuplicateExpression() | 运算符两边的表达式相同 |
| void checkUnreachableCode() | 执行不到的代码 |
| void checkSignOfUnsignedVariable() | 无符号变量的迹象 |
| void checkSuspiciousSemicolon() | 可疑的分号 |
| void checkInvalidFree();  void invalidFreeError(const Token \*tok, bool inconclusive) | 无效内存位置的free()操作 |
| void checkRedundantCopy() | 冗余拷贝：缓冲路径写入之前旧的内容已经被使用 |
| void checkNegativeBitwiseShift() | 有负的右操作数的移位操作 |
| void checkIncompleteArrayFill() | 被menset和类似功能不完全填充的缓冲区 |
| void checkVarFuncNullUB() | 可变函数不使用NULL |
| void checkPipeParameterSize() | 检查pipe()的参数；需要二维整数数组作为参数 |
| void checkCastIntToCharAndBack() | 避免返回值转换成字符型然后又转换成整数型 |
| void checkComparisonFunctionIsAlwaysTrueOrFalse() | 比较函数的值总是 true或false. |
| void checkIgnoredReturnValue() | 被忽略的返回值 |
| void checkRedundantPointerOp() | 冗余的指针操作 |
| void checkLibraryMatchFunctions() | 警告未配置的函数调用 |
| void checkInterlockedDecrement() | InterlockedDecrement()之后非连锁访问的竞争条件 |
| void checkUnusedLabel() | 未使用的标签 |

**checkclass**

|  |  |
| --- | --- |
| void checkExplicitConstructors() | 检查只有参数的构造函数是否使用explicit抑制隐式转换 |
| void privateFunctions() | 检测private函数是否都被调用 |
| void checkMemset()  void checkMemsetType(const Scope \*start, const Token \*tok, const Scope \*type, bool allocation, std::list<const Scope \*> parsedTypes) | 检测memset函数是否合法 |
| void operatorEq() | 检测“=”是否有返回值，以及其返回值是否为常量 |
| void operatorEqRetRefThis() | 检测“=”的返回值是否与“this”有关 |
| void operatorEqToSelf() | 检测“=”是否赋值给自己 |
| void virtualDestructor() | 检测基类的构造函数是否为虚函数 |
| void thisSubtraction() | 检测是否将this->x写成this-x |
| void checkConst() | 检测成员函数是否为常量 |
| void initializerListOrder() | 检测初始化程序参数列表 |
| void checkSelfInitialization() | 检测自己初始化的成员变量 |
| checkPureVirtualFunctionCall() | 检测纯虚的函数是否被调用 |
| checkDuplInheritedMembers() | 检测继承成员变量是否复制 |

**checkexceptionsafety**

|  |  |
| --- | --- |
| void destructors() | 检测是否在析构函数中抛出错误 |
| void deallocThrow() | 检测抛出错误前是否回收内存 |
| void checkRethrowCopy() | 检测是否有错误被重复抛出 |
| void checkCatchExceptionByValue() | 检测是否错误是不是catch的引用而不是值 |
| void nothrowThrows() | 检测是否有不必要的抛出 |
| void unhandledExceptionSpecification() | 检测是否有未特别处理的抛出 |

**checkfunctions**

|  |  |
| --- | --- |
| void checkProhibitedFunctions() | 检测是否使用了禁止的函数 |
| void invalidFunctionUsage() | 检测不合法的函数使用（不合法输入、重复输出等） |
| void checkIgnoredReturnValue() | 检测忽略的返回值 |
| void checkMathFunctions() | 检测没有意义的数学函数 |
| void checkLibraryMatchFunctions() | 检测为声明的函数 |

**checkinternal（该类不是目标代码的检测，而是cppcheck的内部检测）**

|  |  |
| --- | --- |
| void checkTokenMatchPatterns() | 检测是否在Token::Match或Token::findmatch中使用了简单的匹配 |
| void checkTokenSimpleMatchPatterns() | 检测是否在Token::simpleMatch或Token::findsimplematch中使用了复杂的匹配 |
| void checkMissingPercentCharacter() | 检测Token::Match的匹配是否以%结尾 |
| void checkUnknownPattern() | 检测是否有不合法的匹配 |
| void checkRedundantNextPrevious() | 检测多余的Token::next(),Token::previous()和Token::tokAt() |
| void checkExtraWhitespace() | 检测匹配的首尾有无留白 |

**Checkleakautovar 无独特方法**

**checkassert检查断言中是否存在带有副作用的陈述**

|  |  |
| --- | --- |
| voidassertWithSideEffects() | 带副作用的断言 |
| voidsideEffectInAssertError() | 检查带副作用的断言错误 |
| voidcheckVariableAssignment() | 检查是否有副作用发生在变量赋值之前 |
| voidinSameScope() | 判断是否在一个范围内，即使一个返回是在相同的范围内，分配可能不会影响它 |

**checkautovariables检查自动变量**

|  |  |
| --- | --- |
| bool isPtrArg() | 判断是否为指针实参 |
| bool isArrayArg() | 判断是否为数组实参 |
| boolisRefPtrArg() | 判断是否为引用指针实参 |
| bool isNonReferenceArg() | 判断是否为非引用实参 |
| bool isAutoVar() | 判断是否为自动变量 |
| bool isAutoVarArray() | 判断是否为自动变量数组 |
| boolcheckRvalueExpression() | 验证我们是否真的采取了本地变量的地址 |
| bool variableIsUsedInScope() | 判断变量是否在同范围内使用 |
| voidassignFunctionArg() | 检查分配方法实参 |
| voidautoVariables() | 检查自动变量 |
| voidreturnPointerToLocalArray() | 检查返回本地数组的指针 |
| voiderrorReturnAddressToAutoVariable() | 报告自动变量的返回地址错误 |
| void errorReturnPointerToLocalArray() | 报告本地数组的返回指针错误 |
| voiderrorAutoVariableAssignment() | 报告自动变量赋值错误 |
| voiderrorReturnAddressOfFunctionParameter() | 报告函数参数的返回地址错误 |
| void errorUselessAssignmentArg() | 报告无用赋值实参错误 |
| boolreturnTemporary() | 判断是否返回临时变量 |
| boolastHasAutoResult() | 判断语法树是否有自动结构 |
| void returnReference() | 检查返回引用错误 |
| voiderrorReturnReference() | 报告返回引用错误 |
| voiderrorReturnTempReference() | 报告返回临时引用错误 |
| voiderrorInvalidDeallocation() | 报告无效的释放错误 |

**checkBool检查可疑的Bool用法**

|  |  |
| --- | --- |
| void checkComparisonOfFuncReturningBool() | 检查函数返回bool类型的比较 |
| void checkComparisonOfBoolWithBool() | 检查bool类型的变量之间的比较 |
| void checkIncrementBoolean() | 检查对bool类型变量使用后缀增量 |
| void checkComparisonOfBoolWithInt() | 检查一个bool和一个非零的可疑的比较（非1）值 |
| void checkAssignBoolToPointer() | 赋值bool类型值给指针 |
| void checkAssignBoolToFloat() | 赋值bool类型值给float类型 |
| void checkBitwiseOnBoolean() | 检查在位表达中使用bool |
| void checkComparisonOfBoolExpressionWithInt() | 检查布尔表达式与非0或1的整数的比较 |
| void pointerArithBool(); | 检查如果（P + 1）”等有人忘了引用，有人使用指针变量 |
| void pointerArithBoolCond(const Token \*tok); | 检查如果（P + 1）”等有人忘了引用，有人使用指针变量 |

**checkboost检查Boost库使用**

|  |  |
| --- | --- |
| void checkBoostForeachModification() | 检查容器修改而使用boost\_foreach宏 |

**checkstl 检查stl各容器的错误使用,其中一共检验以下错误或使用不当**

|  |  |
| --- | --- |
| stlOutOfBounds(); | 检查容器边界错误 |
| iterators() | 检查迭代器使用错误 |
| mismatchingContainers() | 检查容器不匹配错误 |
| erase() | 检查erase的错误使用 |
| pushback() | 检查push\_back和insert的错误使用 |
| stlBoundaries() | 检查错误的边界条件 |
| if\_find() | 检查查找的使用错误 |
| string\_c\_str() | 检查使用string::c\_str()时的常见错误 |
| checkAutoPointer() | 检查指针的使用和传递问题 |
| uselessCalls() | 检查无用的声明 |
| checkDereferenceInvalidIterator() | 检查不同类型迭代器的错误使用 |

# Cppcheck-1.70版本缺陷分析

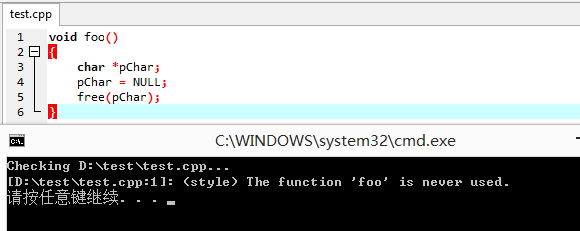
根据软件缺陷的表现特征将已有C与C++语言软件缺陷模式大致为两类：

第一类缺陷有：存储器泄露模型路径敏感、类成员存储器泄露模型不敏感、资源泄露模型路径敏感、空指针变量模型路径敏感、空指针表达式模型路径敏感、数组越界模型路径敏感、非法计算模型路径敏感、未初始化变量模型路径敏感、未初始化成员变量模型路径敏感、死循环结构模型路径敏感、悬挂空指针路径敏感。

第二类缺陷有：缓冲区溢出模型、资源泄露模型。

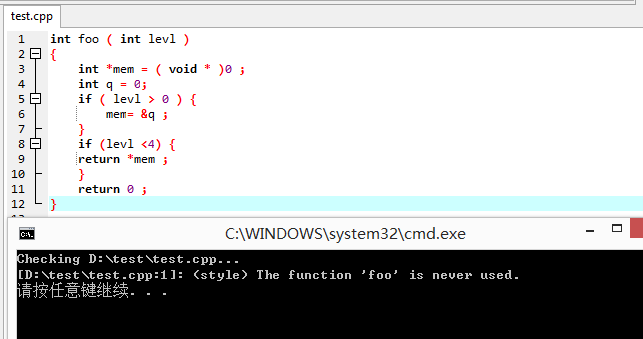
通过收集、归纳、总结的C与C++语言软件缺陷模式并利用Cppcheck对收集的数据进行测试和对比，目前得出以下缺陷Cppcheck仍无法检测出：

**空指针被释放**



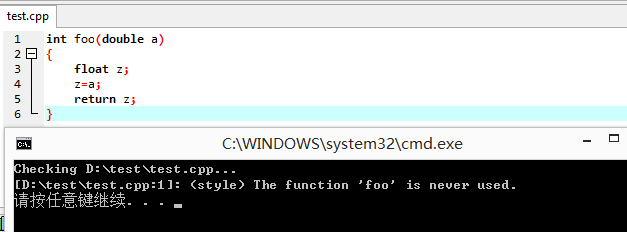
**条件判断语句导致空指针**

例：指针mem在levl大于0的情况下赋值不为空，但是在小于4包括小于0的情况下，mem的取值可能为空，所以会产生空指针错误。



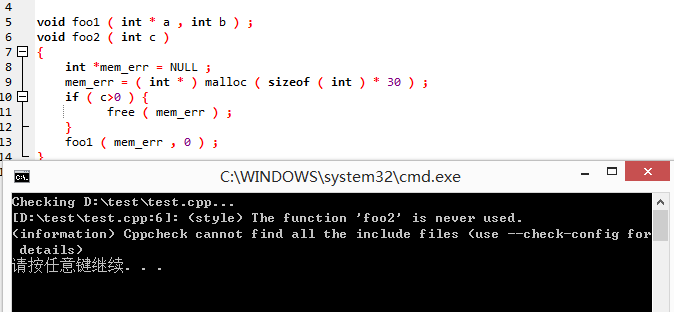
**返回值的类型不一致**

例：函数foo为整型，返回值z是浮点型。

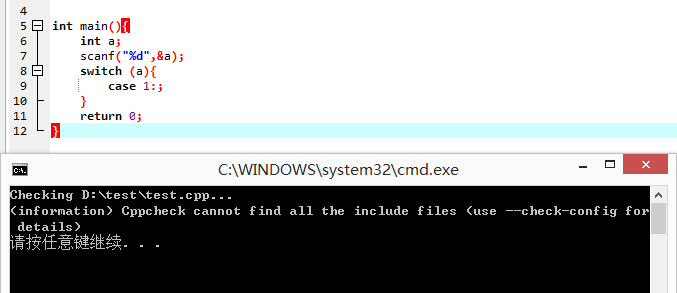


**函数使用已经被释放的指针作为参数**

例：通过c的条件判断来确定是否指针mem\_err分配的内存被释放，在后面的程序中没有对其是否被释放进行判断，而起直接对释放的指针作为形参传递给函数foo1，所以程序会产生被释放的指针作为函数参数缺陷。

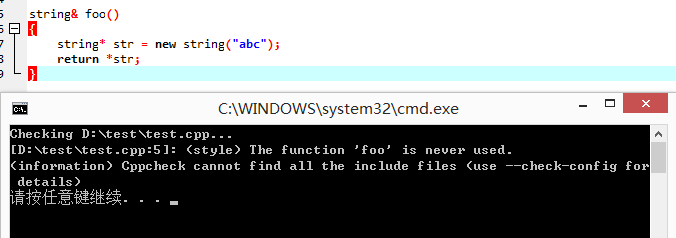


**Switch语句中case条件后没有执行的语句**



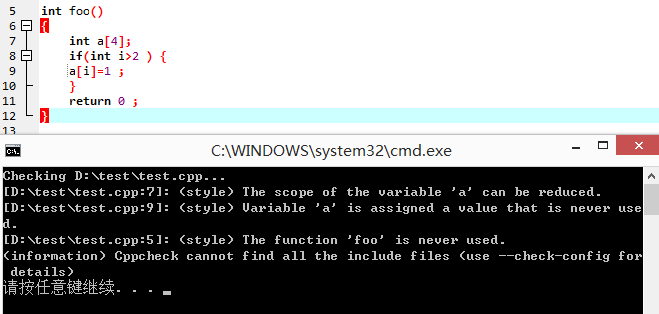
**函数返回new分配的内存的引用**

可能造成内存泄露



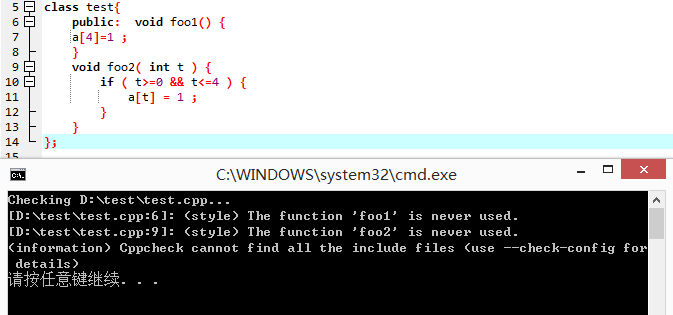
**条件判断导致数组边界越界**

例：条件判断i的值有可能大于3，所以会产生越界中断，越界缺陷的产生取决于条件的判断。

****

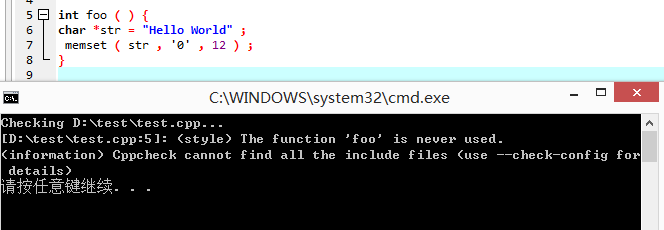
**复杂结构数组边界越界**

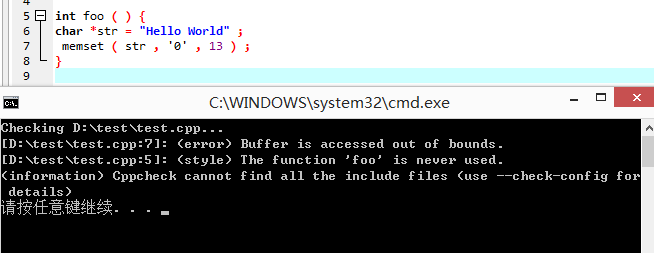
例：成员变量是数组的情况下，在对其进行索引访问时越界。

****

**内存操作函数memset，memcpy引起的缓冲区溢出**

例：memset、memcpy函数需要进行初始化的数组长度小于其函数调用中指明参数的大小。str的字符串长度为11，但是在用memset对其进行赋值的时候的长度大于最大长度，所以产生程序异常。（测试中发现赋值的长度为12时无法检测，为13及以上时却可以检测。）

****

****

# 参考文献

<http://www.doc88.com/p-2075378903086.html>

1. Dead Pointer Usage: when pointer alias local variable that has gone out of scope. ([Cppcheck 1.67 release note](https://sourceforge.net/p/cppcheck/news/2014/10/cppcheck-167/))

   示例代码：

   |  |
   | --- |
   | int \*pi;  {  int i = 0;  pi = &i;  }  (\*pi) = 1; //pi指向的i超出了i的作用域，pi为Dead Pointer，  //此时调用getValueTokenDeadPointer()可获取为其赋值的’&i’的首token’&’ |

   [↑](#footnote-ref-1)