开源静态分析

笔记本： C++
创建时间： 2016/1/7 17:01 **更新时间：** 2016/1/26 15:40
作者： Barret Ren
URL： http://www.cnblogs.com/itech/archive/2011/09/13/2174762.html

引言

最近在项目中使用了静态程序分析工具PC-Lint， 体会到它在项目实施中带给开发人员的方便。PC-Lint是一款针对C/C++语言、windows平台的静态分析工具，FlexeLint是针对其他平 台的PC-Lint版本。由于PC-Lint/FlexeLint是商业的程序分析工具，不便于大家对其进行学习和使用，因而下面我将介绍一个针对C语言 的开源程序静态分析工具——splint。

静态程序分析

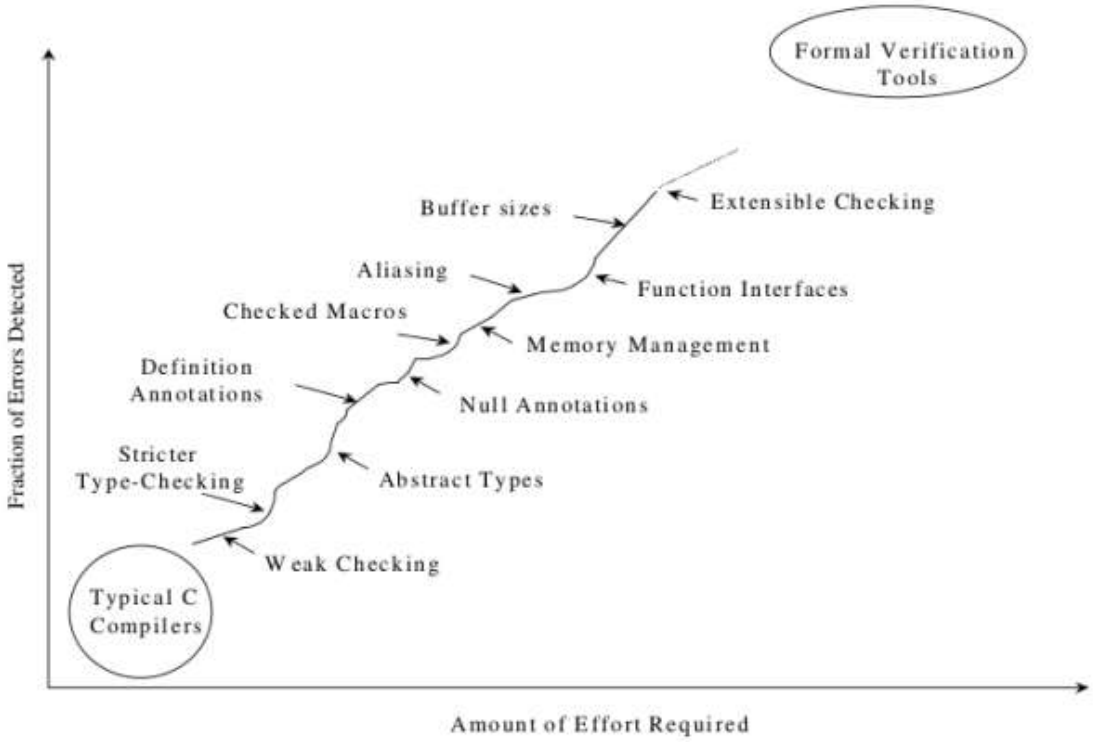
先来说说什么是“静态程序分析(Static program analysis)”，静态程序分析是指使用自动化工具软件对程序源代码进行检查，以分析程序行为的技术，应用于程序的正确性检查、安全缺陷检测、程序优 化等。它的特点就是不执行程序，相反，通过在真实或模拟环境中执行程序进行分析的方法称为“动态程序分析(Dynamic program analysis)”。

那在什么情况下需要进行静态程序分析呢？静态程序分析往往作为一个多人参与的项目中代码审查过程的一个阶段，因编写完一部分代码之后就可以进行静态 分析，分析过程不需要执行整个程序，这有助于在项目早期发现以下问题：变量声明了但未使用、变量类型不匹配、变量在使用前未定义、不可达代码、死循环、数 组越界、内存泄漏等。

静态分析工具在代码通过编译之后再对代码进行分析。我们会问：静态分析工具与编译器相比，所做的工作有什么不同？静态分析工具相比 编译器，对代码进行了更加严格的检查，像数组越界访问、内存泄漏、使用不当的类型转换等问题，都可以通过静态分析工具检查出来，我们甚至可以在分析工具的 分析标准里定义代码的编写规范，在检测到不符合编写规范的代码时抛出告警，这些功能都是编译器没有的。

既然静态分析工具发挥了不小的作用，何不在编译器里兼备静态分析的功能？对于这个问题，S. C. Johnson（他是最古老的静态分析工具Lint的作者）在其1978年发表的论文《Lint, a C Program Checker》中 给出了他的答案：“Lint与C编译器在功能上的分离既有历史原因，也有现实的意义。编译器负责把C源程序快速、高效地转变为可执行文件，不对代码做类型 检查（特别是对分别编译的程序），有益于做到快速与高效。而Lint没有“高效”的要求，可以花更多时间对代码进行更深入、仔细的检查。”

针对空指针提取、未定义变量使用、类型转换、内存管理、函数接口定义等，我们可以在静态分析工具里制定不同的检测标准，以下曲线图说明了在使用splint进行分析时，检测标准与splint运行的开销所对应的关系，从另一个角度看，也说明了静态分析工具与编译器的关系：



splint Linux下安装配置

1、 打开终端；

2、 解压缩：tar zxvfsplint-3.1.2.src.tgz

3、 安装到/usr/local/splint目录下：

sudo mkdir /usr/local/splint

cd splint-3.1.2

./configure --prefix=/usr/local/splint

sudo apt-getinstall flex

make ——注意源代码有问题，要手动添加一个yywarp函数的实现（直接返回return 1即可）

sudo make install

4、 配置，打开当前的bash资源文件

vi ~/.bashrc

在文件最后加入：

exportLARCH_PATH=/usr/local/splint/share/splint/lib

exportLCLIMPORTDIR=/usr/local/splint/share/splint/imports

exportPATH=/usr/local/splint/bin:$PATH

退出文件

source ~/.bashrc

splint

掌握了"静态分析"等概念之后，我们再来看splint。

在Linux命令行下，splint的使用很简单，检测文件*.c，只要这样使用就可以了：

```
splint *.c
```

1.splint消息

我们通过以下例子来认识典型的splint告警信息：

```
1 //splint_msg.c
2 int func_splint_msg1(void)
```

```
 3 {
 4  int a;
 5  return 0;
 6 }
 7 int func_splint_msg2(void)
 8 {
 9  int* a = (int*)malloc(sizeof(int));
10     a = NULL;
11  return 0;
12 }
```

运行splint splint_msg.c之后，我们来看输出的告警信息：

```
splint_msg.c: (in function func_splint_msg1)
splint_msg.c:4:6: Variable a declared but not used
  A variable is declared but never used. Use /*@unused@*/ in front of
  declaration to suppress message. (Use -varuse to inhibit warning)

splint_msg.c: (in function func_splint_msg2)
splint_msg.c:10:2: Fresh storage a (type int *) not released before
assignment:
                       a = NULL
  A memory leak has been detected. Storage allocated locally is not
released
  before the last reference to it is lost. (Use -mustfreefresh to inhibit
  warning)
   splint_msg.c:9:37: Fresh storage a created

Finished checking --- 2 code warnings
```

蓝色字体部分：给出告警所在函数名，在函数的第一个警告消息报告前打印；

红色字体部分：消息的正文，文件名、行号、列号显示在的警告的正文前；

黑色字体部分：是有关该可疑错误的详细信息，包含一些怎样去掉这个消息的信息；

绿色字体部分：给出格外的位置信息，这里消息给出了是在哪里申请了这个可能泄露的内存。

2.检查控制

splint提供了三种方式可进行检查的控制，分别是.splintrc配置文件、flags标志和格式化注释。

flags:splint支持几百个标志用来控制检查和消息报告，使用时标志前加'+'或'-'，'+'标志开启这个标志，'-'表示关闭此标志，下面例子展示了flags标志的用法：

```
splint -showcol a.c    //在检测a.c时，告警消息中列数不被打印
splint -varuse  a.c    //在检测a.c时，告警消息中未使用变量告警不被打印
```

.splintrc配置文件:在使用源码安装splint之后，.splintrc文件将被安装在主目录下，.splintrc文件中对一些标志作了默认的设定，命令行中指定的flags标志会覆盖.splintrc文件中的标志。

格式化注释：格式化注释提供一个类型、变量或函数的格外的信息，可以控制标志设置，增加检查效果，所有格式化注释都以/*@开始，@*/结束，比如在函数参数前加/*@null@*/，表示该参数可能是NULL，做检测时，splint会加强对该参数的值的检测。

3.检测分析内容

1.解引用空指针(Null Dereferences)

在Unix操作系统中，解引用空指针将导致我们在程序运行时产生段错误(Segmentation fault)，一个简单的解引用空指针例子如下：

```c
//null_dereferences.c
int func_null_dereferences(void)
{
 int* a = NULL;
 return *a;
}
```

执行splint null_dereference.c命令，将产生以下告警消息：

```
null_dereference.c: (in function func_null_dereferences)
null_dereference.c:5:10: Dereference of null pointer a: *a
  A possibly null pointer is dereferenced.  Value is either the result of
a
  function which may return null (in which case, code should check it is
not
  null), or a global, parameter or structure field declared with the null
  qualifier. (Use -nullderef to inhibit warning)
   null_dereference.c:4:11: Storage a becomes null

Finished checking --- 1 code warnin
```

2.类型(Types)

我们在编程中经常用到强制类型转换，将有符号值转换为无符号值、大范围类型值赋值给小范围类型，程序运行的结果会出无我们的预料。

```c
//types.c
void splint_types(void)
{
 short a = 0;
 long b = 32768;
    a = b;
 return;
}
```

执行splint types.c命令，将产生以下告警消息：

```
types.c: (in function splint_types)
types.c:6:2: Assignment of long int to short int: a = b
  To ignore type qualifiers in type comparisons use +ignorequals.

Finished checking --- 1 code warning
```

3.内存管理(Memory Management)

C语言程序中，将近半数的bug归功于内存管理问题，关乎内存的bug难以发现并且会给程序带来致命的破坏。由内存释放所产生的问题，我们可以将其分为两种：

- 当尚有其他指针引用的时候，释放一块空间

```c
//memory_management1.c
void memory_management1(void)
{
 int* a = (int*)malloc(sizeof(int));
 int* b = a;
    free(a);
 *b = 0;
 return;
}
```

在上面这个例子中，指针a与b指向同一块内存，但在内存释放之后仍对b指向的内容进行赋值操作，我们来看splint memory_management1.c的结果：

```
memory_management1.c: (in function memory_management1)
memory_management1.c:7:3: Variable b used after being released
  Memory is used after it has been released (either by passing as an only
param
  or assigning to an only global). (Use -usereleased to inhibit warning)
   memory_management1.c:6:7: Storage b released
memory_management1.c:7:3: Dereference of possibly null pointer b: *b
  A possibly null pointer is dereferenced.  Value is either the result of
a
  function which may return null (in which case, code should check it is
not
  null), or a global, parameter or structure field declared with the null
  qualifier. (Use -nullderef to inhibit warning)
   memory_management1.c:5:11: Storage b may become null

Finished checking --- 2 code warnings
```

检查结果中包含了两个告警，第一个指出我们使用了b指针，而它所指向的内存已被释放；第二个是对解引用空指针的告警。

- 当最后一个指针引用丢失的时候，其指向的空间尚未释放

```
1 //memory_management2.c
2 void memory_management2(void)
3 {
4  int* a = (int*)malloc(sizeof(int));
5      a = NULL;
6  return;
7 }
```

这个例子中内存尚未释放，就将指向它的唯一指针赋值为NULL，我们来看splint memory_management2.c的检测结果：

```
memory_management2.c: (in function memory_management2)
memory_management2.c:5:2: Fresh storage a (type int *) not released before
assignment:
                  a = NULL
  A memory leak has been detected. Storage allocated locally is not
released
  before the last reference to it is lost. (Use -mustfreefresh to inhibit
  warning)
   memory_management2.c:4:37: Fresh storage a created

Finished checking --- 1 code warning
```

splint抛出一个告警：类型为int*的a在进行a = NULL赋值前没有释放新分配的空间。

4.缓存边界(Buffer Sizes)

splint会对数组边界、字符串边界作检测，使用时需要加上+bounds的标志，我们来看下面的例子：

```
1 //bounds1.c
2 void bounds1(void)
3 {
```

```
4  int a[10];
5     a[10] = 0;
6  return;
7 }
```

使用splint +bounds bounds1.c命令对其进行检测，结果如下：

```
bounds1.c: (in function bounds1)
bounds1.c:5:2: Likely out-of-bounds store: a[10]
    Unable to resolve constraint:
    requires 9 >= 10
     needed to satisfy precondition:
    requires maxSet(a @ bounds1.c:5:2) >= 10
  A memory write may write to an address beyond the allocated buffer. (Use
  -likelyboundswrite to inhibit warning)

Finished checking --- 1 code warning
```

告警消息提示数组越界，访问超出我们申请的buffer大小范围。再看一个例子：

```
 1 //bounds2.c
 2 void bounds2(char* str)
 3 {
 4  char* tmp = getenv("HOME");
 5  if(tmp != NULL)
 6     {
 7         strcpy(str, tmp);
 8     }
 9  return;
10 }
```

不对这个例子进行详细检查，可能我们不能发现其中隐含的问题，执行splint +bounds bounds2.c之后，会抛出如下告警：

```
bounds2.c: (in function bounds2)
bounds2.c:7:3: Possible out-of-bounds store: strcpy(str, tmp)
    Unable to resolve constraint:
    requires maxSet(str @ bounds2.c:7:10) >= maxRead(getenv("HOME") @
    bounds2.c:4:14)
     needed to satisfy precondition:
    requires maxSet(str @ bounds2.c:7:10) >= maxRead(tmp @ bounds2.c:7:15)
     derived from strcpy precondition: requires maxSet(<parameter 1>) >=
    maxRead(<parameter 2>)
  A memory write may write to an address beyond the allocated buffer. (Use
  -boundswrite to inhibit warning)

Finished checking --- 1 code warning
```

告警消息提示我们：在使用strcpy(str, tmp)进行字符串复制时，可能出现越界错误，因为str的大小可能不足以容纳环境变量“HOME”对应的字符串。绿色字体的内容指示了如何消除告警消息。

CppCheck

1，CppCheck安装

首先从这里下载linux版本的：
http://sourceforge.net/projects/cppcheck/files/cppcheck/

然后下载对应的版本，解压，之后安装：
编译：

make CFGDIR=~/.cppcheck ——需要制定cfg目录不然默认编译会无法使用
安装：

make install
安装后：
将源码目录下的cfg/std.cfg复制到上面的~/.cppcheck目录下面以供使用

2，CppCheck基本使用

格式：cppcheck -j 3 --enable=all src/

1.检查规则：
   1.默认：--enable=error
   2.--enable=all
   3.--enable=unusedFuntion path
   4.--enable=style

2.规则定义：
   1.error：出现的错误
   2.warning：为了预防bug防御性编程建议信息
   3.style：编码格式问题（没有使用的函数、多余的代码等）
   4.portablity：移植性警告。该部分如果移植到其他平台上，可能出现兼容性问题
   5.performance：建议优化该部分代码的性能
   6.information：一些有趣的信息，可以忽略不看的。

3.-j 3 使用3个线程，如果代码工程太大，可以使用15-20个，自己随意发挥，不过还是参考cpuinfo