# 5. Инструменты статического и динамического анализа кода

Разделы:

- Введение
- Статический анализ кода
- Примеры (*cppcheck*)
- Динамический анализ кода
- Примеры (*valgrind*)
- Профилирование
- Примеры (*gprof*)
- Покрытие кода
- Примеры (*gcov*)

# Статический анализ кода

- Под «*статическим анализом*» мы понимаем автоматические методы исследования свойств программного кода без его фактического выполнения
- Свойства, которые мы рассматриваем, включают те, что приводят к преждевременному прекращению или плохо определенным результатам работы
- Они не включают синтаксические ошибки, простые ошибки типов, а также ошибки, связанные с функциональной корректностью программы и т.д.
- Не предназначен для замены тестирования

# Статический анализ кода

- Неправильное управление ресурсами
- Неверные операции
- Мертвый код и данные
- Неполный код
- Незавершение программы, неперехваченные исключения, условия гонок и т.д.

# Статический анализ кода

1. если у программы есть конкретное свойство, анализатор, как правило, имеет возможность сделать вывод лишь о том, что «программа может иметь свойство»
2. если у программы нет конкретного свойства, то появляется шанс определить, что:
   a) наш анализатор на самом деле в состоянии вывести это (т.е. у программы нет свойства)
   b) анализатор может прийти к выводу, что программа может обладать свойством, а может и не обладать – **false positive**

# Статический анализ кода

- Подавление *false positives* может приводить к **false negatives**
- Это необнаруженный реально существующий в коде дефект
  - Если анализ слишком оптимистичен, то он делает неоправданные предположения о влиянии тех или иных операций
  - Если анализ является неполным, т.к. не принимает во внимание все возможные пути выполнения в программе

# Статический анализ кода

- **Потокозависимый анализ** учитывает граф потока управления программы
- **Потоконезависимый** – нет учитывает
- **Путезависмый анализ** рассматривает только допустимые пути внутри программы
- **Путенезависимый** анализ учитывает все пути выполнения, даже те, которые не достигаются никогда
- **Контекстно-зависимый анализ** (межпроцедурный) включает информацию о контексте, например, глобальные переменные и фактические параметры вызова функции при анализе функции
- **Контекстно-независимый анализ** (внутрипроцедурный) не включает информацию о контексте

# Статический анализ кода

- Неразрешимость свойств времени выполнения, подразумевает, что невозможно произвести анализ, который всегда бы находил все дефекты и не производил *false positives*
- Средства статического анализа (фреймворки) называют «**точными**» (или **консервативными** или **безопасными**), если они сообщают обо всех проверяемых дефектах
  - нет *false negatives*, но могут быть *false positives*
- Большинство фреймворков для статического анализа направлены на полноту, пытаясь избежать чрезмерной отчетности ложных срабатываний
  - Большинство современных коммерческих систем (например, *Coverity* и *Klocwork* K7) не являются полными (т.е. они не найдут все фактические дефекты), а также обычно дают ложные срабатывания

# Инструменты (статический анализ)

- Для языков C/C++:
  - **lint* (*adlint*, *splint*);
  - *cppcheck* (http://cppcheck.sourceforge.net/)
  - *viva64*;
  - и др.
- Инструменты для Java:
  - *JLint*;
  - *FindBugs*

# Инструменты (статический анализ)

```
drum@linux-d7fp:~/Teaching/static_analysis> cppcheck --version
Cppcheck 1.66
drum@linux-d7fp:~/Teaching/static_analysis> cat sample1.c
#include <stdio.h>

int main()
{
  FILE* fp = fopen("kuku", "r");

  return 0;
}
drum@linux-d7fp:~/Teaching/static_analysis> cppcheck --std=c99 sample1.c
Checking sample1.c...
[sample1.c:7]: (error) Resource leak: fp
drum@linux-d7fp:~/Teaching/static_analysis> cat sample2.c
#include <stdio.h>

int main()
{
  int* fp = malloc(sizeof (int));

  return 0;
}
drum@linux-d7fp:~/Teaching/static_analysis> cppcheck --std=c99 sample2.c
Checking sample2.c...
[sample2.c:7]: (error) Memory leak: fp
...
```

# Инструменты (статический анализ)

```
...
drum@linux-d7fp:~/Teaching/static_analysis> cat sample3.c
#include <stdio.h>
int main()
{
  int* fp = malloc(sizeof (int));
  free(fp);
  free(fp);
  return 0;
}
drum@linux-d7fp:~/Teaching/static_analysis> cppcheck --std=c99 sample3.c
Checking sample3.c...
[sample3.c:7]: (error) Memory pointed to by 'fp' is freed twice.
[sample3.c:7]: (error) Deallocating a deallocated pointer: fp
[sample3.c:7]: (error) Uninitialized variable: fp
drum@linux-d7fp:~/Teaching/static_analysis> cat sample4.c
#include <stdio.h>
int main()
{
  int a = 0;
  int f = 1/a;
  printf("%d\n", f);
  return 0;
}
drum@linux-d7fp:~/Teaching/static_analysis> cppcheck --std=c99 sample4.c
Checking sample4.c...
[sample4.c:6]: (error) Division by zero.
...
```

# Инструменты (статический анализ)

```
...
drum@linux-d7fp:~/Teaching/static_analysis> cat sample5.c
#include <stdio.h>
#include <stdlib.h>
int main()
{
  int* a = (int*)NULL;
  int f = 1 / *a;
  printf("%d\n", f);
  return 0;
}
drum@linux-d7fp:~/Teaching/static_analysis> cppcheck --std=c99 sample5.c
Checking sample5.c...
[sample5.c:7]: (error) Null pointer dereference
drum@linux-d7fp:~/Teaching/static_analysis> cat sample6.c
#include <stdio.h>
#include <stdlib.h>
int main()
{
  int a[10] = { NULL };
  printf("%d\n", a[10]);
  return 0;
}
drum@linux-d7fp:~/Teaching/static_analysis> cppcheck --std=c99 sample6.c
Checking sample6.c...
[sample6.c:8]: (error) Array 'a[10]' accessed at index 10, which is out of
   bounds.
drum@linux-d7fp:~/Teaching/static_analysis>
```

# Динамический анализ кода

- Динамический анализ (далее – ДА) кода – анализ программного обеспечения, выполняемый при помощи запуска программ на реальном или виртуальном процессоре (в отличие от СА)
- Инструменты ДА обнаруживают программные ошибки в коде, запущенном на исполнение
  - Разработчик имеет возможность наблюдать или диагностировать поведение приложения во время его исполнения, в идеальном случае – непосредственно в целевой среде
  - Во многих случаях в инструменте ДА производится модификация исходного или бинарного кода приложения, чтобы установить ловушки, или **процедуры-перехватчики** (*hooks*), для проведения инструментальных измерений

# Динамический анализ кода

- Плюсы ДА:
  1. Редко возникают *false positives* – высокая продуктивность по нахождению ошибок
  2. Для отслеживания причины ошибки может быть произведена полная трассировка стека и среды исполнения
  3. Захватываются ошибки в контексте работающей системы
- Минусы ДА:
  1. Происходит вмешательство в поведение системы в реальном времени
     - степень вмешательства зависит количества используемых инструментальных вставок
     - Это не всегда приводит к возникновению проблем, но об этом нужно помнить при работе с критическим ко времени кодом
  2. Полнота анализа ошибок зависит от степени покрытия кода
     - Кодовый путь, содержащий ошибку, должен быть обязательно пройден, а в контрольном примере должны создаваться необходимые условия для создания ошибочной ситуации

# Динамический анализ кода

- Пример инструмента ДА – *Valgrind*
  - http://www.valgrind.org/
- A number of useful tools are supplied as standard
  - Memcheck is a memory error detector. It helps you make your programs, particularly those written in C and C++, more correct
  - Cachegrind is a cache and branch-prediction profiler
    - It helps you make your programs run faster
  - Callgrind is a call-graph generating cache profiler
    - It has some overlap with Cachegrind, but also gathers some information that Cachegrind does not
  - Helgrind is a thread error detector
    - It helps you make your multi-threaded programs more correct
  - DRD is also a thread error detector
    - It is similar to Helgrind but uses different analysis techniques and so may find different problems
  - Massif is a heap profiler
    - It helps you make your programs use less memory
  - DHAT is a different kind of heap profiler
    - It helps you understand issues of block lifetimes, block utilisation, and layout inefficiencies
  - etc

# Динамический анализ кода

- Список проверок памяти (*Memcheck*), которые можно выполнить с помощью Valgrind:
  - Использование неинициализированной памяти
  - Утечки памяти
  - Переполнения памяти
  - Повреждение стека
  - Использование указателей памяти после того, как соответствующая память была освобождена
  - Несоответствующие указатели в *malloc*/*free*

# Инструменты (статический анализ)

```
drum@linux-d7fp:~/Teaching/dynamic_analysis> valgrind --version
valgrind-3.10.0
drum@linux-d7fp:~/Teaching/dynamic_analysis> cat valg1.c
#include <stdio.h>
int main()
{
  int* p;
  int c = *p;
  if (0 == c)
  {
    printf("Kuku!!!\n");
  }
  return 0;
}
drum@linux-d7fp:~/Teaching/dynamic_analysis> gcc -g valg1.c -o valg1
drum@linux-d7fp:~/Teaching/dynamic_analysis> valgrind ./valg1
==10058== Memcheck, a memory error detector
==10058== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==10058== Using Valgrind-3.10.0 and LibVEX; rerun with -h for copyright info
==10058== Command: ./valg1
==10058==
==10058== Use of uninitialised value of size 4
==10058==    at 0x804844A: main (valg1.c:6)
==10058==
==10058==
==10058== HEAP SUMMARY:
==10058==     in use at exit: 0 bytes in 0 blocks
==10058==   total heap usage: 0 allocs, 0 frees, 0 bytes allocated
==10058==
==10058== All heap blocks were freed -- no leaks are possible
==10058==
==10058== For counts of detected and suppressed errors, rerun with: -v
==10058== Use --track-origins=yes to see where uninitialised values come from
==10058== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
```

# Инструменты (статический анализ)

```
drum@linux-d7fp:~/Teaching/dynamic_analysis> cat valg2.c
#include <stdio.h>
#include <stdlib.h>
int main()
{
  int* i = (int *) malloc(sizeof (int));
  *i = 10;
  free(i);
  printf("%d\n", *i);
  return 0;
}
drum@linux-d7fp:~/Teaching/dynamic_analysis> gcc -g valg2.c -o valg2
drum@linux-d7fp:~/Teaching/dynamic_analysis> valgrind ./valg2
==10092== Memcheck, a memory error detector
==10092== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==10092== Using Valgrind-3.10.0 and LibVEX; rerun with -h for copyright info
==10092== Command: ./valg2
==10092==
==10092== Invalid read of size 4
==10092==    at 0x80484E0: main (valg2.c:9)
==10092==  Address 0x4206028 is 0 bytes inside a block of size 4 free'd
==10092==    at 0x402AA3D: free (in /usr/lib/valgrind/vgpreload_memcheck-x86-linux.so)
==10092==    by 0x80484DB: main (valg2.c:8)
==10092==
10
==10092==
==10092== HEAP SUMMARY:
==10092==     in use at exit: 0 bytes in 0 blocks
==10092==   total heap usage: 1 allocs, 1 frees, 4 bytes allocated
==10092==
==10092== All heap blocks were freed -- no leaks are possible
==10092==
==10092== For counts of detected and suppressed errors, rerun with: -v
==10092== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
drum@linux-d7fp:~/Teaching/dynamic_analysis>
```

# Инструменты (статический анализ)

```
drum@linux-d7fp:~/Teaching/dynamic_analysis> cat valg3.c
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
void get_strings(char const* in)
{
  char* cmd;
  int len = strlen("strings: ") + strlen(in) + 1;
  cmd = malloc(len);
  snprintf(cmd, len, "strings %s", in);
  if (0 != system(cmd))
    fprintf(stderr, "Smth wrong %s.\n", cmd);
  free(cmd);
  free(cmd);
}
int main(int argc, char* argv[])
{
  get_strings(argv[0]);
}
drum@linux-d7fp:~/Teaching/dynamic_analysis> gcc -g valg3.c -o valg3
...
```

# Инструменты (статический анализ)

```
...
drum@linux-d7fp:~/Teaching/dynamic_analysis> valgrind ./valg3
==10108== Memcheck, a memory error detector
==10108== Copyright (C) 2002-2013, and GNU GPL'd, by Julian Seward et al.
==10108== Using Valgrind-3.10.0 and LibVEX; rerun with -h for copyright info
==10108== Command: ./valg3
==10108==
/lib/ld-linux.so.2
libc.so.6
_IO_stdin_used
strlen
malloc
stderr
system
fprintf
__libc_start_main
snprintf
free
__gmon_start__
GLIBC_2.0
PTRh
[^_]
strings %s
Smth wrong %s.
;*2$"

...
```

# Инструменты (статический анализ)

```
...

==10108== Invalid free() / delete / delete[] / realloc()
==10108==    at 0x402AA3D: free (in /usr/lib/valgrind/vgpreload_memcheck-x86-linux.so)
==10108==    by 0x8048603: get_strings (valg3.c:14)
==10108==    by 0x804861B: main (valg3.c:19)
==10108==  Address 0x4206028 is 0 bytes inside a block of size 17 free'd
==10108==    at 0x402AA3D: free (in /usr/lib/valgrind/vgpreload_memcheck-x86-linux.so)
==10108==    by 0x80485F8: get_strings (valg3.c:13)
==10108==    by 0x804861B: main (valg3.c:19)
==10108==
==10108==
==10108== HEAP SUMMARY:
==10108==     in use at exit: 0 bytes in 0 blocks
==10108==   total heap usage: 1 allocs, 2 frees, 17 bytes allocated
==10108==
==10108== All heap blocks were freed -- no leaks are possible
==10108==
==10108== For counts of detected and suppressed errors, rerun with: -v
==10108== ERROR SUMMARY: 1 errors from 1 contexts (suppressed: 0 from 0)
drum@linux-d7fp:~/Teaching/dynamic_analysis>
```

# Профилирование кода

- Важно иметь точную информацию о том, где именно расходуется время в программе на реальных входных данных, если мы желаем эффективно оптимизировать код
- Такие действия называются *профилированием кода*
- Можно выполнять ручное профилирование, но предпочтительнее поручить эту работу какому-либо автоматизированному средству
- Существует достаточно большое количество программ для профилирования кода
- Мы будем использовать GNU-профайлер (*gprof*)

# Профилирование в жизненном цикле

- В общем случае код создается для решения следующих задач:
  - *Корректность работы программного обеспечения*
  - *Удобство в обслуживании программного обеспечения*
  - *Производительность программного обеспечения* (Здесь можно использовать профилировщики)

# Использование gprof

- Использование *gprof* несложно
- Это выполнение трех шагов:
  1. Разрешить добавлять информацию для профилировщика при компиляции программного кода
  2. Выполнить программный код для того, чтобы создать профильные данные
  3. Запустить *gprof* с передачей ему файла с профильными данными

# Использование gprof

- Создает файл, который содержит результаты анализа в читабельной форме
- Помимо иной информации, также содержит две таблицы
- **Простой профиль** содержит обзор временной информации по работе функций
- **Граф вызовов** показывает все вызовы функций, которые привели к данной функции, какие функции вызываются из данной функции и т.д.

# Использование gprof

```
/* test_gprof.c */
#include<stdio.h>
void new_func1(void);
void func1(void)
{
    printf("\n Inside func1 \n");
    int i = 0;
    for(;i<0xffffffff;i++);
    new_func1();
    return;
}
static void func2(void)
{
    printf("\n Inside func2 \n");
    int i = 0;
    for(;i<0xfffffaa;i++);
    return;
}
```

# Использование gprof

```
int main(void)
{
    printf("\n Inside main()\n");
    int i = 0;
    for(;i<0xffffff;i++);
    func1();
    func2();
    return 0;
}

/* test_gprof_new.c */
#include<stdio.h>
void new_func1(void)
{
    printf("\n Inside new_func1()\n");
    int i = 0;
    for(;i<0xffffffee;i++);
    return;
}
```

# Использование gprof

-pg : Generate extra code to write profile information suitable for the analysis program gprof. You must use this option when compiling the source files you want data about, and you must also use it when linking.

# Использование gprof

```
$ gcc -Wall -pg test_gprof.c test_gprof_new.c -o test_gprof

$ ls
test_gprof  test_gprof.c  test_gprof_new.c
$ ./test_gprof
...
$ ls
gmon.out  test_gprof test_gprof.c  test_gprof_new.c
$

$ gprof test_gprof gmon.out > analysis.txt

$ ls
analysis.txt  gmon.out  test_gprof  test_gprof.c  test_gprof_new.c
```

# Использование gprof

```
Flat profile:

Each sample counts as 0.01 seconds.
  %   cumulative   self              self     total
 time   seconds   seconds    calls   s/call   s/call  name
 33.83      7.98     7.98        1     7.98     7.98  func2
 33.06     15.78     7.80        1     7.80    15.59  func1
 33.02     23.57     7.79        1     7.79     7.79  new_func1
  0.08     23.59     0.02                             main

 %         the percentage of the total running time of the
time       program used by this function.

cumulative a running sum of the number of seconds accounted
 seconds   for by this function and those listed above it.

 self      the number of seconds accounted for by this
seconds    function alone.  This is the major sort for this
           listing.

calls      the number of times this function was invoked, if
           this function is profiled, else blank.

 self      the average number of milliseconds spent in this
```

# Использование gprof

```
ms/call    function per call, if this function is profiled,
          else blank.

 total     the average number of milliseconds spent in this
ms/call    function and its descendents per call, if this
          function is profiled, else blank.

name       the name of the function.  This is the minor sort
           for this listing. The index shows the location of
          the function in the gprof listing. If the index is
          in parenthesis it shows where it would appear in
          the gprof listing if it were to be printed.

                     Call graph (explanation follows)

granularity: each sample hit covers 4 byte(s) for 0.04% of 23.59 seconds
```

# Использование gprof

```
index % time    self  children    called     name
                                                 <spontaneous>
[1]    100.0    0.02   23.57                 main [1]
                7.80    7.79       1/1           func1 [2]
                7.98    0.00       1/1           func2 [3]
-----------------------------------------------
                7.80    7.79       1/1           main [1]
[2]     66.1    7.80    7.79       1         func1 [2]
                7.79    0.00       1/1           new_func1 [4]
-----------------------------------------------
                7.98    0.00       1/1           main [1]
[3]     33.8    7.98    0.00       1         func2 [3]
-----------------------------------------------
                7.79    0.00       1/1           func1 [2]
[4]     33.0    7.79    0.00       1         new_func1 [4]
-----------------------------------------------
```

This table describes the call tree of the program, and was sorted by the total amount of time spent in each function and its children.

Each entry in this table consists of several lines. The line with the index number at the left hand margin lists the current function. The lines above it list the functions that called this function, and the lines below it list the functions this one called.

# Использование gprof

```
This line lists:
     index          A unique number given to each element of the table.
          Index numbers are sorted numerically.
          The index number is printed next to every function name so
          it is easier to look up where the function in the table.

     % time         This is the percentage of the `total' time that was spent
          in this function and its children.  Note that due to
          different viewpoints, functions excluded by options, etc,
          these numbers will NOT add up to 100%.

     self This is the total amount of time spent in this function.

     children       This is the total amount of time propagated into this
          function by its children.

     called         This is the number of times the function was called.
          If the function called itself recursively, the number
          only includes non-recursive calls, and is followed by
          a `+' and the number of recursive calls.

     name The name of the current function.  The index number is
          printed after it.  If the function is a member of a
          cycle, the cycle number is printed between the
          function's name and the index number.
```

# Использование gprof

```
For the function's parents, the fields have the following meanings:

    self This is the amount of time that was propagated directly
         from the function into this parent.

    children       This is the amount of time that was propagated from
         the function's children into this parent.

    called         This is the number of times this parent called the
         function `/' the total number of times the function
         was called.  Recursive calls to the function are not
         included in the number after the `/'.

    name This is the name of the parent.  The parent's index
         number is printed after it.  If the parent is a
         member of a cycle, the cycle number is printed between
         the name and the index number.

If the parents of the function cannot be determined, the word
`<spontaneous>' is printed in the `name' field, and all the other
fields are blank.
```

# Использование gprof

```
For the function's children, the fields have the following meanings:

     self This is the amount of time that was propagated directly
          from the child into the function.

     children       This is the amount of time that was propagated from the
          child's children to the function.

     called         This is the number of times the function called
          this child `/' the total number of times the child
          was called.  Recursive calls by the child are not
          listed in the number after the `/'.

     name This is the name of the child.  The child's index
          number is printed after it.  If the child is a
          member of a cycle, the cycle number is printed
          between the name and the index number.

If there are any cycles (circles) in the call graph, there is an
entry for the cycle-as-a-whole.  This entry shows who called the
cycle (as parents) and the members of the cycle (as children.)
The `+' recursive calls entry shows the number of function calls that
were internal to the cycle, and the calls entry for each member shows,
for that member, how many times it was called from other members of
the cycle.
```

# Использование gprof

```
Index by function name

   [2] func1                   [1] main
   [3] func2                   [4] new_func1
```

# Кастомизация вывода gprof

- Можно подавить вывод статических функций.

```
$ gprof -a test_gprof gmon.out > analysis.txt
```

- Когда большого количества информации не требуется можно ее подавить с помощью флага *-b*.

```
$ gprof -b test_gprof gmon.out > analysis.txt
```

- В том случае, когда нам достаточно только простого профиля, мы можем использовать флаг *-p*. Его рекомендуется использовать совместно с флагом *-b*.

```
$ gprof –p -b test_gprof gmon.out > analysis.txt
```

# Кастомизация вывода gprof

- Вывод информации по конкретной функции в простом профиле выполняется указанием имени функции с опцией *-p*.

```
$ gprof –pfunc2 -b test_gprof gmon.out >
    analysis.txt
```

- Если не требуется информация о простом профиле, то ее вывод может быть подавлен указанием опции *-P*:

```
$ gprof -P -b test_gprof gmon.out > analysis.txt
```

# Кастомизация вывода gprof

- Можно организовать вывод простого профиля, за исключением некоторой функции. Для этого так же используется опция *-P* совместно с именем функцию, которую нужно исключить из вывода.

```
$ gprof –Pfunc2 -b test_gprof gmon.out >
    analysis.txt
```

- Можно выводить информацию о графе вызовов с использованием опции *-q*:

```
gprof -q -b test_gprof gmon.out > analysis.txt
```

# Кастомизация вывода gprof

- Можно выводить в графе вызовов информацию только об определенной функции. Это выполняется указанием имени функции после флага *-q*.

```
$ gprof –qfunc2 -b test_gprof gmon.out >
    analysis.txt
```

- Если вообще не требуется информация о графе вызовов, то следует использовать опцию *-Q*.

```
$ gprof -Q -b test_gprof gmon.out > analysis.txt
```

# Кастомизация вывода gprof

- При выводе файла аналитики будет показана только информация о простом профиле. Так же можно подавить вывод информации в графе вызовов об определенной функции путем указания имени функции вместе с флагом *-Q*.

```
$ gprof –Qfunc2 -b test_gprof gmon.out >
    analysis.txt
```

# Кастомизация вывода gprof

- Мы можем получить листинг «аннотированного исходного кода», в котором выводится исходный код приложения с отметками о количестве вызовов каждой функции
- Для использования этой возможности нужно откомпилировать исходный код с разрешенной отладочной информацией, для того чтобы исходный код был помещен в исполняемый файл:

```
$ gprof –A test_gprof gmon.out > analysis.txt
```

# gcov

- Coverage analysis shows you how much different code branches get used
  - This is useful to see if all code is covered by tests and to identify performance problems
  - Since compile and run times are significantly increased by these flags, you may want to only use these flags inside a given project directory or even just one *.*c* file (e.g. *myfile.c*)
- $ *gcc -g -Oo -fprofile-arcs -ftest-coverage  mycode.c -o mycode*
- $*./mycode*
  - files *mycode.gcda* and/or *mycode.gcno* will be created
- $ *gcov mycode.gcda*
  - file *mycode.c.gcov* will be created (and sometime *.*h.gcov*)
- $ *cat mycode.c.gcov*
- There are three states the execution count can have:
  - ""-"" is for lines without code
  - ""#####"" are lines which never got executed
  - A number tells us how often that line got executed, watch out for optimizations, which can distort the value of blocks

# gcov

```
/* GCOV example */
#include <stdio.h>

int main(void)
{
  int i;

  for (i = 1; i < 10; ++i)
  {
    if (i % 3 == 0)
      printf("%d is divisible by 3\n", i);
    if (i % 11 == 0)
      printf("%d is divisible by 11\n", i);
  }

  return 0;
}
```

# Инструментарий и источники - CA

- Lint man page - http://www.unix.com/man-page/FreeBSD/1/lint
- FindBugs - http://findbugs.sourceforge.net/
- JLint - http://jlint.sourceforge.net/
- Klocwork TruePath - http://www.klocwork.com/products/insight/klocwor
  /
- Coverity SAVE - http://www.coverity.com/products/coverity-save.ht
- List of tools for static code analysis - http://en.wikipedia.org/wiki/List_of_tools_for_static

# Инструментарий и источники - ДА

- Dynamic program analysis - http://en.wikipedia.org/wiki/Dynamic_program_ana
- Использование статического и динамического анализа для повышения качества продукции и эффективности разработки - http://www.swd.ru/print.php3?pid=828
- Выявление ошибок работы с памятью при помощи valgrind - http://www.opennet.ru/base/dev/valgrind_memory
- PurifyPlus - http://unicomsi.com/products/purifyplus/
- Intel Inspector XE - https://software.intel.com/en-us/intel-inspector-xe
- Clang Static Analyzer - http://clang-analyzer.llvm.org/

# Некоторые профилировщики и средства покрытия кода

- GNU Binutils - http://www.gnu.org/software/binutils/
- Sysprof, System-wide Performance Profiler for Linux - http://sysprof.com/
- OProfile - A System Profiler for Linux (News) - http://oprofile.sourceforge.net/
- VTune Amplifier XE by Intel Corporation - http://software.intel.com/en-us/intel-vtune-amplifier-xe
- Java Profiler - JProfiler - http://www.ej-technologies.com/products/jprofiler/overv
- List of performance analysis tools - http://en.wikipedia.org/wiki/List_of_performance_analys
- Gcov docs - http://gcc.gnu.org/onlinedocs/gcc/Gcov.html