C++ libre de Bugs

Cómo podemos encontrar algunos bugs en nuestros proyectos de C++

Luis Felipe Domínguez Vega

lfdominguez@nauta.cu

ldominguezvega@gmail.com

H3R3T1C, 2016

Contenido

Introduciéndonos en el tema

En nuestros días de andanza con C++ nos hemos encontrado con los famosos `Segmentation Fault`, `std::terminate`, etc. errores que no siempre ocurren por "problemas de la PC", sino que como humanos que somos, cometemos errores en nuestra codificación. C++ es un lenguaje de propósito general donde se nos permite crear casi cualquier software. Pero las propias reglas de codificación nos permiten introducir bugs en nuestros códigos, los cuales ni "fijándonos" bien a veces los vemos. De ahí que existan herramientas para detectarlos, el primero de ellos por supuesto es el propio compilador, pero por problemas de velocidad de compilación a veces no se revisan aspectos del código. Para ello existen los analizadores estáticos como cppcheck, cpplint, etc. En los siguientes epígrafes intentaré abarcar los temas más generales sobre estos analizadores.

Compilando nuestro código

Si bien todos utilizamos, para ahorrar mucho trabajo, los sistemas de autoconstrucción, como ✝ Autotools ✝, ♡ CMake ♡, QMake, etc. a veces compilamos como buenos Guru que somos utilizando directamente el comando del compilador.

Usando g++ (GCC) o clang++ (Clang)

La cantidad de opciones con que cuenta GCC para la compilación se va completamente del propósito de este documento por lo que se mostrará como compilar un fichero simple de C++. Supongamos que tenemos el siguiente código en un fichero main.cpp:

```cpp
#include <iostream>

void method () {
    enum TEST {CASE1, CASE2};
    TEST test;

    switch (test) {
    case CASE1: break;
    }
}

int main() {
    method();
    int *var = new int;
    *var = 41;
    var = new int;
    *var = 42;

    std::cout << "La solución al Universo es: " << *var << std::endl;

    return 0;
}
```

Para compilarlo y enlazarlo, quedando en un ejecutable llamado "main"

```
g++ main.cpp -o main
```

```
clang++ main.cpp -o main
```

Tratando con las advertencias (warnings)

Los compiladores cuentan con una bandera (flag) para modificar el tratamiento de los errores y advertencias en el código, para nuestro caso aconsejo utilizar `-Wall -Werror`, por pasos:

- `-Wall`: Habilita toda una serie de advertencias que vienen desactivadas por defecto.
- `-Werror`: Trata las advertencias como errores, muchas personas no toman en serio las advertencias, pero en lo personal las tomo como posibles bugs.

En el código de ejemplo si lo compilamos con g++ sin esas flags no se "queja" de advertencias, pero pruébenlo con dichas flags y se darán cuenta del cambio. En este caso los errores (advertencias, pero como tiene `-Werror` los trata como errores) que traería consigo son:

1. En el switch no se tratan todos los valores del enum, eso puede ser un error de que al programador se le haya olvidado incluirlo como un caso.
2. La variable utilizada en el switch "test" no ha sido inicializada, por lo que puede contener valor basura.

```
main.cpp: In function ‘void method()’:
main.cpp:7:9: error: enumeration value ‘CASE2’ not handled in switch [-Werror=switch]
  switch (test) {
         ^
main.cpp:7:2: error: ‘test’ is used uninitialized in this function [-Werror=uninitialized]
  switch (test) {
  ^
cc1plus: all warnings being treated as errors
```

En el caso de `clang++` sin las banderas anterirores encuentra la advertencia 1, pero no la 2, requieriendo dicha bandera.

Para el caso de los sistemas de autoconstrucción, podemos indicarlo en:

- ☥ Autotools ☥: En los `Makefile.am` que contengan objetivos binarios o bibliotecas

```
<objetivo>_CPPFLAGS = -Wall -Werror
```

- ♡ CMake ♡: En los `CMakeLists.txt` que contengas objetivos binarios o bibliotecas

```
SET( CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -Wall -Werror" )
```

- QMake: En este caso puede ser o bien en el .pro (en lo personal la mejor opción) o a la hora de ejecutar `qmake` con la variable `QMAKE_CXXFLAGS`:

```
qmake QMAKE_CXXFLAGS+="-Wall -Werror"
```

Hay una bandera muy específica que es `-pedantic` donde se habilitan comprobaciones específicas del lenguaje, pero es demasiado "pedante" y aunque a veces la aconsejo es más bien para mantener la compatibilidad con la norma del código de C++.

Los sanitize

Los compiladores modernos implementan un mecanismo de corrección de errores en modo runtime llamado sanitize, en este caso comento acerca de "leak" y "undefined" para detectar leaks de memoria y comportamientos indefinidos respectivamente. Si se dieron cuenta en el código de ejemplo, la creación de la variable con memoria dinámica es completamente intencional, ahí se puede ver un problema de fuga de memoria, pues no se libera explícitamente antes de asignar nuevamente la memoria. Dicho problema no es reconocido por las banderas anteriormente descritas. Muchas personas utilizan "valgrind" u otros programas para la detección de estos errores, pero con el método que les comento a continuación se ahorran la disminución drástica de tiempo de ejecución, debido a que con estas banderas lo que hace el compilador es generar código explícitamente para detectar estos errores. Pero nos preguntamos entonces, de que por qué si esto es tan bueno para los programas, por qué no siempre utilizarlo y compilarlo así; pues estos mecanismos relentizan mucho nuestros programas ya que tendrían que realizar muchas más comprobaciones e instrucciones, por lo que se recomienda utilizarlo en modo de desarrollo y quitarlo en modo de producción; no siendo así para las banderas de advertencias.

Los argumentos serían:

- `-fsanitize=leak`: Habilita el modo de detección de fugas de memoria.
- `-fsanitize=undefined`: Habilita el modo de detección de comportamiento indefinido.
- `-fno-omit-frame-pointer`: No borra la dirección en el fichero fuente donde se encuentra una función.
- `-g`: Para generar información de depuración.

Para habilitar dichos componentes `-fsanitize=leak -fsanitize=undefined -fno-omit-frame-pointer -g`:

- Manualmente: Como buen Guru

```
g++ -Wall -Werror -fsanitize=leak -fsanitize=undefined -fno-omit-frame-pointer
-g main.cpp -o main
```

```
clang++ -Wall -Werror -fsanitize=leak -fsanitize=undefined
-fno-omit-frame-pointer -g main.cpp -o main
```

- ✝ Autotools ✝: En los `Makefile.am` que contengan objetivos binarios o bibliotecas

```
<objetivo>_CPPFLAGS = -Wall -Werror -fsanitize=leak -fsanitize=undefined
-fno-omit-frame-pointer -g
```

- ♡ CMake ♡: En los `CMakeLists.txt` que contengas objetivos binarios o bibliotecas

```
SET( CMAKE_CXX_FLAGS "${CMAKE_CXX_FLAGS} -Wall -Werror -fsanitize=leak
-fsanitize=undefined -fno-omit-frame-pointer -g" )
```

- QMake: En este caso puede ser o bien en el .pro (en lo personal la mejor opción) o a la hora de ejecutar `qmake` con la variable `QMAKE_CXXFLAGS` y `QMAKE_LFLAGS`:

```
qmake QMAKE_CXXFLAGS+="-Wall -Werror -fsanitize=leak -fsanitize=undefined
-fno-omit-frame-pointer -g" QMAKE_LFLAGS+="-fsanitize=leak
-fsanitize=undefined"
```

Ahora ejecutamos nuestro programa, verificando la consola por algun error de "undefined" como la llamada a un puntero nulo por ejemplo. Los "leak" de memoria siempre son mostrados al cerrar nuestro programa, se arrojan todos en la consola, indicando la función, el fichero y la línea de código donde se encuentra.

```
=================================================================
==20913==ERROR: LeakSanitizer: detected memory leaks

Direct leak of 4 byte(s) in 1 object(s) allocated from:
    #0 0x7f3859cee436 in operator new(unsigned long) (/usr/lib/x86_64-linux-gnu/liblsan.so.0+0xd436)
    #1 0x400b1c in main /tmp/main.cpp:14
    #2 0x7f38583bf60f in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x2060f)

Direct leak of 4 byte(s) in 1 object(s) allocated from:
    #0 0x7f3859cee436 in operator new(unsigned long) (/usr/lib/x86_64-linux-gnu/liblsan.so.0+0xd436)
    #1 0x400b57 in main /tmp/main.cpp:16
    #2 0x7f38583bf60f in __libc_start_main (/lib/x86_64-linux-gnu/libc.so.6+0x2060f)

SUMMARY: LeakSanitizer: 8 byte(s) leaked in 2 allocation(s).
```

Podríamos probar agregando a nuestro código lo siguiente:

```
1 ++var;
2 var = 0x0000;
3 *var = *var;
```

Y compilen nuevamente, verán como les sale un error en tiempo de ejecución diciendo el fichero y la línea de código donde se derreferencia el puntero nulo, esto es por el sanitize de "leak".

```
La solución al Universo es: 42
main.cpp:23:13: runtime error: load of null pointer of type 'int'
fish: './main' terminated by signal SIGSEGV (Address boundary error)
```

Utilizando scan-build de clang

Clang trae consigo un conjunto de herramientas aparte del propio compilador. Dichas herramientas son para diferentes tipos de tareas, como la de "modernizar" el código (convirtiéndo nuestro código para que se ajuste a los nuevos estándares), reformateo del código, etc. En este caso comentaré acerca de `scan-build`, es una herramienta que utiliza por debajo a clang para compilar y habilita un argumento en clang para realizar análisis de código más profundo, además de elaborar todo un conjunto de ficheros HTML y al finalizar genera el `index.html` para acceder a todos estos errores. Esta aplicación contiene una serie analizadores para los cuales aconsejo habilitar los que muestro en la ejecución del comando. Se utiliza como "reemplazo" de `make`, por ejemplo, si se usaba

```
make -j4
```

ahora se debería usar

```
scan-build -o CarpetaDeSalida --html-title "Título del reporte" -maxloop 10
-analyzer-config stable-report-filename=true -enable-checker
alpha.core.BoolAssignment -enable-checker
alpha.core.CallAndMessageUnInitRefArg -enable-checker alpha.core.CastSize
-enable-checker alpha.core.CastToStruct -enable-checker
alpha.core.DynamicTypeChecker -enable-checker alpha.core.FixedAddr
-enable-checker alpha.core.IdenticalExpr -enable-checker
alpha.core.PointerArithm -enable-checker alpha.core.PointerSub -enable-checker
alpha.core.SizeofPtr -enable-checker alpha.core.TestAfterDivZero
-enable-checker alpha.cplusplus.VirtualCall -enable-checker
```

```
alpha.deadcode.UnreachableCode -enable-checker alpha.security.ArrayBoundV2
-enable-checker alpha.security.MallocOverflow -enable-checker
alpha.security.ReturnPtrRange -enable-checker
alpha.security.taint.TaintPropagation -enable-checker alpha.unix.Chroot
-enable-checker alpha.unix.PthreadLock -enable-checker alpha.unix.SimpleStream
-enable-checker alpha.unix.Stream -enable-checker
alpha.unix.cstring.BufferOverlap -enable-checker
alpha.unix.cstring.NotNullTerminated -enable-checker
alpha.unix.cstring.OutOfBounds -enable-checker
nullability.NullableDereferenced -enable-checker
nullability.NullablePassedToNonnull -enable-checker
nullability.NullablePassedToNonnull -enable-checker optin.performance.Padding
-enable-checker security.FloatLoopCounter -enable-checker
security.insecureAPI.rand -enable-checker security.insecureAPI.strcpy make -j4
```

Analizando sus argumentos, sería:

- `-o`: Carpeta donde se ubicarán los reportes, dentro de dicha carpeta genera una con la fecha de ejecución donde se almacena realmente el reporte.
- `--html-title`: Título que tendrá el sitio.
- `--maxloop`: Cantidad de pasadas a las sentencias repetitivas (`for` , `while` , etc.) para detectar errores.
- `-analyzer-config`: Parámetros que se le pasan al analizador de clang.
- `-enable-checker`: Habilita un analizador.

Lo interesante de `scan-build` es que los errores lo identifica con una serie de pasos lógicos, es decir como los detectó, ejemplo: