AED: Herramientas para C++

gabriel.spranger@utec.edu.pe
macarena.oyague@utec.edu.pe

Junio 2022

Índice

1. Introducción

Cuando uno desarrolla software, se apoya de varias herramientas que van a minimizar ciertos dolores. Por ejemplo, existen *formatters* que automáticamente *formatean* el código para que tenga un estilo consistente y ya no hayan más disputas por cómo usar las "{}", si usar *tabs* o espacios, etc; los *linters* analizan el código de manera *estática*, es decir sin la necesidad de correr el código, y nos reportan errores de *performance*, *bugs* que podrían ocurrir, etc; *hooks* que nos permiten hacer ciertas acciones cuando ocurren ciertos eventos de `git` que más que nada se usan para asegurar un estilo consistente de mensajes de *commit* y asegurar que el código esté *formateado*, *linteado* y compile antes de que el *commit* entre en efecto. En este documento, veremos estas tres herramientas poderosas en el contexto de C++.

2. clang-format

Usualmente cuando varias personas trabajan sobre un solo proyecto, resulta que cada una tiene su estilo para programar. Uno podría preferir usar espacios en vez de tabs, otro podría preferir que un tab sea equivalente a dos espacios, mientras que otro podría preferir que equivalga a cuatro, uno podría preferir definir bloques de código así

```
void printHello()
{
    std::cout << "hello\n";
}
```

y otros así

```
void printHello() {
    std::cout << "hello\n";
}
```

y la lista sigue y sigue. Para evitar estas discrepancias y evitar tener que *formatear* el código a mano (lo cual puede tomar mucho tiempo), se podría usar un *formatter*.

Un *formatter* es un programa que recibe como *input* un conjunto de reglas y un conjunto de archivos. Las reglas describen cómo *formatear* el código, por ejemplo: usar tabs, cada tab equivale a dos espacios, poner llaves en una nueva línea y mucho más. Los archivos son los archivos que el *formatter* va a *formatear* de acuerdo al conjunto de reglas.

Hay distintos *formatters* para distintos lenguajes. Por ejemplo, para Javascript el más popular es Prettier (aunque también funciona para HTML, CSS, etc) y para C/C++ es clang-format.

Para clang-format, el archivo donde se definen las reglas es el `.clang-format` el cual está en formato YAML. Un archivo `.clang-format` simple se ve así:

```
Language: Cpp
IndentWidth: 8
```

Le estamos indicando a clang-format que el lenguaje que se va a *formatear* es C++ y que queremos que cada nivel de identación sea equivalente a ocho espacios.

Ahora para correr clang-format, usamos el siguiente comando:

```
clang-format -i -style=file file1.cpp file2.h file3.cpp ...
```

`-i` le indica a clang-format que modifique los archivos si es necesario. `-style=file` le indica a clang-format que las reglas de *formatting* estarán en el archivo `.clang-format`. Al correr el comando, veremos que se *formatean* los archivos de acuerdo a las reglas que se han especificado. Sin embargo, no es buena práctica *tipear* manualmente todos los archivos que queremos *formatear*. Por lo tanto, se puede correr un comando que automatiza esto y busca todos los archivos con las extensiones `.h` y `.cpp` y se los pasa a clang-format:

```
find -E . -regex ".*\.(cpp|h)" -type f -print0 | xargs -0 clang-format -i
    -style=file
```

Uno podría leerse toda la lista de reglas que clang-format tiene e ir habilitando una por una hasta que tenga todas las reglas que desea. Es válido hacerlo. También uno podría simplemente hacer que su `.clang-format` se base en las reglas que organizaciones más grandes utilizan para su código. Por ejemplo, podríamos usar el `.clang-format` que Google usa para Chromium, el que Apple usa para WebKit, etc, todos escritos en C++. Para ello, corremos el siguiente comando:

```
clang-format -style=<style> -dump-config > .clang-format
```

Donde `<style>` puede ser llvm, gnu, google, chromium, microsoft, mozilla o webkit. Con ese comando, se creará un archivo `.clang-format` con las reglas de *formatting* del *style* escogido. Además, podemos modificar este archivo si algunas reglas no nos gustan. Si quieren saber qué significa el `>`, pueden revisar el siguiente link.

Finalmente, es posible configurar VS Code para que apenas guardemos un archivo, VS Code llame al comando clang-format y de acuerdo al archivo .clang-format (si es que existe) formatea el archivo que acabamos de guardar. Si quieren integrar esto a VS Code pueden revisar este link. Si no les funciona, no duden en escribirnos por Discord.

3. clang-tidy

Los *linters* son programas que analizan código de manera estática, es decir, sin correr el código. El propósito de los *linters* es que reporten *bugs* (los que se pueden detectar con análisis estático), malas prácticas, etc. Por ejemplo, en vez de hacer

```
for (auto item : algunVector) {
    cout << item << "\n";
}
```

el *linter* te puede recomendar que hagas esto

```
for (const auto& item : algunVector) {
    cout << item << "\n";
}
```

Ya que dentro del FOR no cambias item (`const`) y es más rápido que se agarre una referencia al `item` en vez de una copia en cada iteración (`&`).

Otro ejemplo es que puede ser que tengas el siguiente código

```
int i = 0, j = 0;
while (i < 10) {
  ++j;
}
```

y te compila bien, pero el *linter* te indica que tienes un `while` infinito dado que no se actualiza la variable dentro de la condición dentro del cuerpo del `while`. Esta regla se llama bugprone-infinite-loop. Pueden ver la lista completa de reglas que el *linter* clang-tidy puede aplicar aquí.

Al igual que clang-format, clang-tidy tiene un archivo donde se especifican las reglas que quieres que se apliquen a tu código; este archivo es `.clang-tidy` y también está en formalo YAML. Un archivo básico de clang-tidy se ve así:

```
Checks: 'performance-*,clang-diagnostic-*'
FormatStyle: file
WarningsAsErrors: ''
CheckOptions:
  - key: performance-for-range-copy.WarnOnAllAutoCopies
    value: 'true'
```

La primera línea (`Checks`) indica que queremos que clang-tidy aplique todas las reglas de la categoría **performance** y **clang-diagnostic**. Estas son todas las categorías que clang-tidy tiene junto con una explicación sobre qué son:

Name prefix	Description
abseil-	Checks related to Abseil library.
altera-	Checks related to OpenCL programming for FPGAs.
android-	Checks related to Android.
boost-	Checks related to Boost library.
bugprone-	Checks that target bug-prone code constructs.
cert-	Checks related to CERT Secure Coding Guidelines.
clang-analyzer-	Clang Static Analyzer checks.
concurrency-	Checks related to concurrent programming (including threads, fibers, coroutines, etc.).
cppcoreguidelines-	Checks related to C++ Core Guidelines.
darwin-	Checks related to Darwin coding conventions.
fuchsia-	Checks related to Fuchsia coding conventions.
google-	Checks related to Google coding conventions.
hicpp-	Checks related to High Integrity C++ Coding Standard.
linuxkernel-	Checks related to the Linux Kernel coding conventions.
llvm-	Checks related to the LLVM coding conventions.
llvmlibc-	Checks related to the LLVM-libc coding standards.
misc-	Checks that we didn't have a better category for.
modernize-	Checks that advocate usage of modern (currently "modern" means "C++11") language constructs.
mpi-	Checks related to MPI (Message Passing Interface).
objc-	Checks related to Objective-C coding conventions.
openmp-	Checks related to OpenMP API.
performance-	Checks that target performance-related issues.
portability-	Checks that target portability-related issues that don't relate to any particular coding style.
readability-	Checks that target readability-related issues that don't relate to any particular coding style.
zircon-	Checks related to Zircon kernel coding conventions.

Figura 1: Todas las reglas (o *checks*) posibles para clang-tidy. Tomado de este aquí.

`FormatStyle: file` le dice a clang-tidy que use las reglas en el archivo `.clang-format` para *formattear* aparte de hacer *linting*. Esto significa que basta con tener un `.clang-format`, un `.clang-tidy` y correr solo el comando `clang-tidy` para hacer *linting* y *formatting*.

`WarningsAsErrors` sirve para especificar qué categorías de reglas o reglas individuales reportarán sus sugerencias como errores. Esto sirve en la parte de Git Hooks cuando queremos abortar el *commit* si el *linter* devolvió un error (si devuevle solo warnings, no se considera con error).

Cada regla (o *check*) revisa ciertas cosas de nuestro código, pero podemos también personalizar estas reglas mediante las opciones que tenga disponible. Por ejemplo, tomemos la regla performance-for-range-copy (aquí pueden ver una lista de todas las reglas que ofrece clang-tidy). Lo que hace dicha regla es que revisa los bucles de la forma `for (auto item : container)` y dependiendo de unas heurísticas, sugiere que se convierta a `for (const auto& item : container)` o `for (auto& item : container)` para evitar que en cada iteración del bucle se haga una copia innecesaria de `item`, lo cual no es eficiente. Si entran al enlace de la regla, podrán ver en la sección **Options** que hay dos: `WarnOnAllAutoCopies` y `AllowedTypes`. `CheckOptions` es una lista de *key-value*s que sirve justamente para usar las opciones que una regla tenga. En el .clang-tidy simple, estamos di-

ciendo que la opción `WarnOnAllAutoCopies` de la regla `performance-for-range-copy` va a ser "true", lo cual de acuerdo a la documentación, dice que es necesario para habilitar la regla y por lo tanto que el *linter* siempre tire un *warning* cuando se hace `for (auto item : container)`.

Al igual que el archivo `.clang-format`, podemos usar el comando `clang-tidy` para que nos genere un archivo `.clang-tidy` con las reglas que queramos:

```
clang-tidy --dump-config -checks="bugprone-*,concurrency-*,
    cppcoreguidelines-*,clang-analyzer-*,modernize-*,performance-*,
    portability-*,readability-*,misc-*" --format-style="file" > .clang-
    tidy
```

En este caso estamos generando un archivo `.clang-tidy` donde queremos los checks de la categoría **bugprone**, **concurrenct**, etc. Además, notar que de una vez le estamos indicando que va a agarrar las reglas de *formatting* de un archivo `.clang-format` ya existente (`--format-style="file"`).

Ahora que ya tenemos el archivo, falta solo correr:

```
find -E . -regex ".*\.(cpp|c|hpp|h)" -type f -print0 | xargs -0 clang-
    tidy -fix
```

El flag `-fix` le indica a clang-tidy que arregle ciertos errores que pueden ser arreglados.

4. git hooks

Hay un concepto llamado **Continuous Integration (CI)** el cual significa establecer ciertos *pipelines* (cuando mencionamos pipeline, piensen en etapas por las cuales el código pasa: etapa 1, etapa 2, etc), por las cuales el código que uno quiere integrar a un repositorio pasarán, para asegurar que el código que se está integrando sea "correcto". "Correcto" puede significar que compile bien (etapa 1), que esté bien *formateado* (etapa 2), que no tenga errores de *lint* (etapa 3), que pase todos los *tests* (etapa 4) y finalmente hacerle *deploy* para que tome efecto en el mundo real (etapa 5).

Hay distintas maneras de implementar estos *pipelines*. Tenemos: GitHub Actions, el cual se configura en un repositorio para que corra (el *pipeline*) cuando ocurren ciertos eventos; git hooks, similar a GitHub Actions, pero ocurre en el *local repo* a diferencia de GitHub Actions que corre en el *remote repo*; etc.

`git` permite que podamos correr *scripts* o recibir *notificaciones* cuando ciertas acciones ocurren en el *local repo*. Los eventos que personalmente nos parecen más importantes y que más hemos visto varias veces son:

- `pre-commit`: Ocurre apenas se corre el comando `git commit`. Usualmente se usa para ver que el código compile bien, correr el *formatter*, correr el *linter* y finalmente correr los *tests*.

- `commit-msg`: Se suelen mantener *convenciones* para manejar mensajes de commit, ya que uno podría poner "avance", "lo intentamos xd", etc, y esto no es fácil de entender por los demás colaboradores del proyecto, ya que podrían estar confundidos por la terminología usada y se verán obligados a preguntarle al autor del *commit* de qué se tratan sus cambios. Además, otros beneficios que trae manejar un formato común para todos los mensajes de *commit* es que se pueden escribir programas que analizan los *commits* de un repositorio y como se conoce la estructura de cada uno (ya que se maneja una convención) se puede interpretar y automatizar la generación de CHANGELOGs y también que cada persona que lee el mensaje del *commit*, sabrá exactamente qué se hizo, en qué modulo y cómo esto impacta al proyecto. Una convención popular es esta y se puede usar el `commit-msg` hook para validar que el mensaje del *commit* se adhiere a un estándar como ese.

5. Ejemplo

Hemos preparado un repositorio template donde verán un buen ejemplo de cómo se usa `clang-tidy`, `clang-format` y `git-hooks`. Tienen que seguir las instrucciones del README para que les funcionen los `git-hooks`.

6. Referencias

- https://en.wikipedia.org/wiki/Lint_(software)
- https://www.clangpowertools.com/blog/getting-started-with-clang-format-style-options.html
- https://clang.llvm.org/extra/clang-tidy/index.html
- https://www.atlassian.com/git/tutorials/git-hooks
- https://snippets.cacher.io/snippet/9d2999b47ebdcf614883

7. Anexos

Para los que tienen macOS, así es cómo habilitan los comandos `clang-format` y `clang-tidy`:

```
ln -s "/usr/local/opt/llvm/bin/clang-format" "/usr/local/bin/clang-format
    "
ln -s "/usr/local/opt/llvm/bin/clang-tidy" "/usr/local/bin/clang-tidy"
```

Si quieren saber qué es el comando `ln` pueden revisar este link.