

Onboarding - Recap 01 -

Collapse context

Proceso de Building y Testing

El proceso de buildeo o *building* es más que compilar, así como *testing* es más que ver que algo funciona. (perdón mi *spanglish*)

En este *recap* recorrerás las distintas fases y verás las mismas herramientas que se usan en Taller (y en la vida profesional)

- Linters
- Compilación y linkeo
- Memory profiler
- Redirecciones en Bash
- Diff & hexdump

Algunas de estas herramientas (linters and static checkers) las podrás incluso llamar a cada commit.

Mas allá de responder el cuestionario es **muy** recomendable que pongas en práctica las herramientas vistas.

Correlas, mira sus outputs, familiarízate con ellas.

Son las mismas que se usaran para **evaluar** tu TP.

Tu respuesta es exitosa! Tu calificación es 100.0%. [Tarea #66c8e5816096f1682c4592d0]

Pregunta 1:

El código C++ pasa por distintas etapas hasta llegar a un ejecutable.

Simplificadamente estas son:

- precompilación
- compilación
- linkeo (o enlazado)

En Taller usaremos `gcc`, la suite de compilación de GNU.

Te será de mucha ayuda familiarizarte con él, con sus configuraciones y con la forma que tiene de comunicar errores

Digamos que el compilador `gcc` emite el siguiente error:

```
a.cpp: In function 'int main()':
a.cpp:6:21: error: too many arguments to function 'int foo(int)
      return foo(12, 8);
              ^
a.cpp:1:5: note: declared here
int foo(int i) {
```

Que clase de error es?

Si tienes dudas o el proceso de compilación te es algo extraño, hichimos un [tutorial explicativo](#) de como funciona *paso a paso*. Vale la pena antes de continuar!

☐ `foo` esta declarada pero no definida y por lo tanto es un error de linkeo

☐ `foo` recibe un `int` pero se le estan pasando 2 y el linker no sabe como enlazar (linker).

☐ Es un error de compilación por que `foo` esta declarada pero no definida.

☐ Es un error de precompilación por que se esta generando antes de compilar.

☒ La firma (signature) de `foo` no corresponde a como se la esta llamando; es un error de compilación.

Pregunta 2:

Y este error, que significa?

```
/usr/bin/ld: /tmp/ccr80e9c.o: in function 'main':
a.cpp:(.text+0xf): undefined reference to 'foo(int, int)'
collect2: error: ld returned 1 exit status
```

Referencias:

- [tutorial explicativo](#) por si aun no lo hiciste :)

☐ Es un error de compilación por que `foo` esta declarada pero no definida.

☒ `foo` esta declarada pero no definida y por lo tanto es un error de linkeo

☐ La librería `ld` no esta disponible por lo tanto es un error de linkeo (`foo` debe estar implementada en la librería `ld`).

Pregunta 3:

Los *static checkers* son programas que buscan potenciales errores en el código mientras que los *linters* se enfocan en inconsistencias de estilos en el código.

Muchas veces un *lint* mismo programa funciona tanto como *static checker* como *linter* y son usados como sinónimos.

En Taller usamos varios, entre ellos `cpplint` y `cppcheck` como parte del proceso de evaluación de las entregas.

Y son herramientas q deberas correr sobre tu código para cazar cualquier bug lo antes posible.

Supone el siguiente código C++

```
int main() { // Línea 1 de a.cpp
    if (1)
    {
        return foo(12, 8);
    }
} // Línea 6 de a.cpp
```

Estos son algunos errores estilísticos que `cpplint` encontró. Como los arreglarías?

```
a.cpp:2: Tab found; better to use spaces [whitespace/tab] [1]
a.cpp:3: Tab found; better to use spaces [whitespace/tab] [1]
a.cpp:3: { should almost always be at the end of the previous
a.cpp:3: Missing space before { [whitespace/braces] [5]
a.cpp:4: Tab found; better to use spaces [whitespace/tab] [1]
a.cpp:5: Tab found; better to use spaces [whitespace/tab] [1]
Done processing a.cpp
Total errors found: 6
```

Hay varias respuestas posibles, no una sola. **Marcar todas!**

☐ Las llaves { deben estar indentadas con un solo espacio y no con varios ni con tabs.

☒ Hay que escribir en la misma linea el `if` con su correspondiente llave y no en líneas separadas. O sea `if (1) {`

☒ Se están usando tabs para la indentación. Hay que remplazarlos por espacios.

Pregunta 4:

`clang-format` puede automáticamente reescribir tu código siguiendo un estilo específico. Con que flag le indicas que quieres reemplazar tu código *in place* con su versión formateada?

☐ Flag `-i` (por reescribir, del ingles *rewrite*).

☒ Flag `-inplace` (del ingles *in place*).

☐ Flag `-r` (del ingles *rewrite*).

☐ Flag `-r` (por reescribir, del ingles *rewrite*).

Pregunta 5:

`clang-format` te resolverá el 99.99% de los errores de estilo de forma automática.

Sin embargo `cpplint` puede aun indicarte algunos errores. Muchos de ellos serán errores reales pero algunos, los de estilo, puede que sean falsos positivos. Digamos que `cpplint` es un poco estricto en este sentido.

Supone el siguiente código C++

```
using namespace taller::aoe::internals;
```

Supone ademas que `cpplint` indica un error de estilo en dicha línea y crees que es un falso positivo. Cómo le indicas a `cpplint` que ignore esa línea del análisis?

Usa *Google*.

☐ Agregando un comentario `// IGNORE` en la misma línea.

☒ Agregando un comentario `// NOCLINT` en la misma línea.

☐ Agregando un comentario `// SUPPRESS` en la misma línea.

Pregunta 6:

`cppcheck` es capaz de detectar errores como *buffer overflows* incluso antes de ejecutar el programa.

Dado el siguiente código C++, que error da `cppcheck` ?

```
int main() {
    char buf[10];
    buf[1000] = 42;
}
```

Nota: **corrre** `cppcheck` en tu entorno de trabajo, sea tu Linux o el docker de desarrollo provisto por Taller. Cuanto más puedas *checkear antes* de subir tus entregas al Sercom, más *oportunidades de aprobar* tendrás!

Es una excelente oportunidad para que pruebes `cppcheck` en tu máquina.

☐ Division by zero.

☐ Array `buf[10]` accessed at index 1000, this may be less performant.

☒ Array `buf[10]` accessed at index 1000, which is out of bounds.

Pregunta 7:

`cppcheck` no es perfecto y hay veces que genera *falsos positivos*.

Como en Taller somos estrictos y pedimos entregas con 0 errores, un falso positivo no te va a permitir entregar.

Si estas seguro que es un falso positivo y no hay un error real, puedes usar una *supresión* (o *suppression*).

Imaginate que `cppcheck` te dice que hay un overflow en el siguiente código:

```
int foo() {
    buf[1000] = 42; // <-- overflow aqui
}
```

Como podrías suprimir el error? Acá *Google is your friend* y si quieres probar localmente cada una de las opciones del multiple choice para ver cual es la correcta, te aviso que tienes que pasarle a `cppcheck` el flag `--inline-suppr`.

☐ Hay que poner un comentario en la primer linea del archivo diga `// cppcheck-suppress`

☐ Hay que poner un comentario arriba de `buf[1000]` que diga `// cppcheck-suppress no-overflow`

☒ Hay que poner un comentario arriba de `buf[1000]` que diga `// cppcheck-suppress arrayIndexOutOfBounds`

Pregunta 8:

`cpplint`, `clang-format` y `cppcheck` pueden ejecutarse de forma automatica en el momento de crearse un commit usando hooks de *pre-commit*. Esto te permite estar todo el tiempo chequeando la calidad de tu código.

Al principio puede ser molesto, pero recuerda que `cpplint` y `cppcheck` estan para detectarte bugs *antes* que te exploten en la cara.

Pero es verdad hay veces que uno quiere commitear *sin* ejecutar los hooks (sin correr los linters). Hay otras veces que uno quiere correr los hooks *ahora* sin necesidad de un commit.

Cuales de estas afirmaciones son **correctas**? Hay varias respuestas posibles, no una sola. **Marcar todas!**

☒ Para crear un commit sin correr los hooks se debe ejecutar `git commit --no-verify`

☒ Para correr `cppcheck` sin necesidad de un commit se debe ejecutar `pre-commit run --all-files cppcheck`

☐ Para correr `cpplint` sin necesidad de un commit se debe ejecutar `git commit --hook=cpplint`

☐ Para crear un commit sin correr los hooks se debe ejecutar `git commit --skip-hooks`

Pregunta 9:

Asi como hay *static checkers* que analizan el código sin ejecutarlo tenemos *runtime checkers* que instrumentan y ejecutan el código para detectar errores en runtime.

`valgrind` es un *memory error detector*: detecta leaks de memoria, buffer overflows, variables sin inicializar y archivos abiertos entre otras cosas.

Es **extraordinariamente** útil y forma parte de los checkeos que corremos en Taller. Y te será una herramienta muy útil. Correla todo lo que puedas.

Se tiene el siguiente output de `valgrind`. Qué significa *cada* error?

```
// Error 1
Conditional jump or move depends on uninitialised value(s)
at 0x109176: main (a.cpp:6)

// Error 2
Open file descriptor 3: foobar
at 0x4C831AE: open (open4.c:48)
by 0x414E51: _IO_file_open (fileops.c:189)
by 0x414EFC: _IO_file_fopen@GLIBC_2.2.5 (fileops.c:281)
by 0x409150: _fopen_internal (iofopen.c:75)
by 0x10915F: main (a.cpp:3)

// Error 3
Open file descriptor 2: /dev/pts/9
<inherited from parent>

// Error 4
Open file descriptor 1: /dev/pts/9
<inherited from parent>

// Error 5
Open file descriptor 0: /dev/pts/9
<inherited from parent>

// Error 6
10 bytes in 1 blocks are definitely lost in loss record 1 of 6
at 0x4B3650F: operator new[](unsigned long)
by 0x10916D: main (a.cpp:5)
```

Referencias:

- [Errores de valgrind](#)
- [Guia de valgrind](#)

Nota: hay mas de una opción correcta. **Marcar todas!**

☐ Errores 3 al 5: el archivo `/dev/pts/9` representa a la terminal donde fue ejecutado el programa y no fue cerrado por el mismo. Falta llamar explícitamente a un `close`.

☐ Errores 3 al 5: son errores que suceden cuando una clase hija no implementa un destructor y no libera los recursos heredados de su clase padre.

☒ Error 6: hay un leak de memoria. Falta llamar a `delete (o delete[])`.

☒ Error 2: se abrió un archivo en `a.cpp` línea 3 que nunca fue cerrado. Falta llamar explícitamente a un `close`.

☒ Errores 3 al 5: son falsos positivos. No son errores realmente.

☐ Error 6: es un falso positivo. Valgrind no siempre puede detectar la liberación de memoria.

☒ Error 1: se esta haciendo `if(foo)` donde `foo` es una variable sin inicializar.

☐ Error 2: es un error de la libreria de C++ (open64.c y fileops.c). No es habitual pero como todo software las librerías no están exentas de bugs. Este es un caso!

Pregunta 10:

Ahora veremos algo de *shell scripting* que lo usaras para probar tus entregas (y nosotros también).

Ademas, es una excelente oportunidad para que practiques y juegues con tu Linux. Ya tienes uno instalado, no?

El programa `wc` cuenta la líneas, palabras y bytes de un archivo que es pasado por **entrada estándar** (en C++ esto lo harías leyendo de `std::cin`).

Cuales de las siguientes formas de ejecución le pasan a `wc` un archivo por **entrada estándar**?

Referencias:

- [Redirecciones y pipe en Bash](#)
- `std::cin` y `std::cout`
- [manpage de wc](#) (opcional) si quieres ver que otras cosas hace `wc`.

☒ `wc < foo.txt`

☐ `wc foo.txt | wc`

☐ Ninguna. `wc` lee de la entrada estandar, esto es, el teclado y no es posible redirigir un archivo a la entrada estandar.

☐ `wc >> foo.txt`

Pregunta 11:

El programa `cat` lee uno o varios archivos por línea de comandos (como parámetros) e imprime por **salida estándar** sus contenidos en orden concatenándolos (en C++ esto lo harías escribiendo en `std::cout`).

Cuales de las siguientes maneras de llamar a `cat` hace que la **salida estándar** quede guardada en un archivo?

Referencias:

- [Redirecciones y pipe en Bash](#)
- `std::cin` y `std::cout`
- [manpage de cat](#) (opcional) si quieres ver que otras cosas hace `cat`.

☐ `cat foo.txt bar.txt 2> baz.txt`

☒ `cat foo.txt bar.txt > baz.txt`

☐ `cat foo.txt bar.txt | baz.txt`

Pregunta 12:

Ahora, la magia.

Podemos **conectar** la salida de un programa a la entrada de otro para realizar procesamientos más complejos, encadenados.

Por ejemplo, si se quiere contar con `wc` la cantidad de líneas, palabras y bytes totales de 3 archivos leídos por `cat`, qué deberías escribir en la consola?

Referencias:

- [Redirecciones y pipe en Bash](#)

☐ `cat f.txt g.txt h.txt >> wc`

☐ `wc < cat f.txt g.txt h.txt`

☒ `cat f.txt g.txt h.txt | wc`

Pregunta 13:

Cuando trabajes con datos binarios, sean archivos o mensajes de red, es mas fácil ver los datos en **hexadecimal**.

`hexdump` (o `hd`) es una de las tantas herramientas en Linux para ello.

Qué es lo que hace el siguiente comando? `hexdump -C foo.bin`

Referencias:

- [manpage de hexdump](#)

☒ Muestra los bytes del archivo en hexadecimal y en ASCII.

☐ Toma los bytes de a 2 y los muestra como números en hexadecimal (números de 2 bytes)

☐ Muestra 3 representaciones del archivo: en base 64 (los primeros 8 dígitos), en hexadecimal y en ASCII

Pregunta 14:

Se ejecutó `hexdump -C foo.bin` y se obtuvo la siguiente salida:

```
00000000  61 62 63 64      |abcd|
00000004
```

Cuales de las siguientes observaciones son correctas? Hay mas de una, **marcarlas todas!**

Referencias:

- [manpage de hexdump](#)
- [Papa noel no existe, el EOF tampoco](#)

☒ La letra `b` en ASCII corresponde al numero 62 en hexadecimal.

☐ El último byte de `foo.bin` es el End of File (EOF)

☐ Los primeros bytes de `foo.bin` son todos ceros.

☐ `foo.bin` es un archivo de 8 bytes.

☐ La letra `b` en ASCII corresponde al numero 64 en hexadecimal.

☒ `foo.bin` es un archivo de 4 bytes.

☒ El último byte de `foo.bin` es el caracter `d`.

☐ El End of File (EOF) es mostrado por `hexdump` como `00000004`.

Pregunta 15:

En Taller tenemos automatizadas las pruebas de ejecución para testear tus entregas.

Simplificadamente lo que hacemos es correr tu programa y capturar su salida en un archivo `obtenido.txt`.

Luego comparamos ese archivo con `esperado.txt`: si son iguales el programa pasó el test, sino, marcamos las diferencias.

La comparación la hacemos usando el programa `diff`.

Suponete que corrimos:

```
diff -u --no-dereference --new-file --ignore-trailing-space --
obtenido.txt 2022-01-07 21:36:18.892000000 +0000
+++ esperado.txt 2022-01-07 21:35:26.856000000 +0000
@@ -1,5 +1,5 @@
I wake up to the sounds of the silence that allows
-For my mind to run with my ear up to the ground
+For my mind to behold the stories that are told
when my back is to the world that was smiling when I turned
@@ -8,9 +9,9 @@
But once you turn they hate us

Oh, the misery
-everybody wants to be my enemy
-spare the sympathy
-everybody wants to be my enemy
+Everybody wants to be my enemy
+Spare the sympathy
+Everybody wants to be my enemy

(Look out for yourself)

@@ -19,4 +19,3 @@
(Look out for yourself)

But I'm ready
-
```

Referencias:

- [Como interpretar la salida de "diff -u"](#)
- [manpage de diff](#)
- [Meld un diff pero gráfico. Muy útil!](#)
- [Bonus track opcional](#)

Cuales de las siguientes observaciones son correctas? Hay mas de una, **marcarlas todas!**

Tomate el tiempo para **entender** la salida de `diff`. Podrás entender los errores marcados por el Sercom más fácilmente.

☒ El primer bloque con diferencias está mostrando la línea 1 y las siguientes 5 líneas.

☐ Hay 2 bloques con diferencias (la salida del `diff` tiene 2 partes)

☒ La palabra *around* **no** esta presente en `obtenido.txt` pero **si** esta presente en `esperado.txt`, de ahí la diferencia.

☐ La palabra *around* **si** esta presente en `obtenido.txt` pero **no** esta presente en `esperado.txt`, de ahí la diferencia.

☒ Hay 3 bloques con diferencias (la salida del `diff` tiene 3 partes)

☐ El segundo bloque con diferencias está mostrando las líneas de la 8 a la 19.

☒ Una de las diferencias es que hay un salto de línea (línea vacía) presente en `obtenido.txt` pero no en `esperado.txt`. O sea, **no** se esta esperando esa línea de más (hay una línea de más en `obtenido.txt`).

☐ Una de las diferencias es que hay un salto de línea (línea vacía) presente en `esperado.txt` pero no en `obtenido.txt`. O sea, **si** se esta esperando un línea más (hay una faltante en `obtenido.txt`).

Pregunta 16:

En Linux todo programa que finaliza termina con un código de retorno o exit code.

Es una muy buena convención que si el programa detecta algun error finalice con un código distinto de 0 mientras que si todo funcionó bien termine con 0.

Para saber cual fue el código de retorno del último programa ejecutado podes hacer `echo "$?"`.

Tene en cuenta que luego de ejecutar el `echo`, si volves a a ejecutar `echo "$?"` estaras viendo el código de retorno del primer `echo`.

Como parte de los tests automáticos que corre el Sercom se verificara el código de retorno.

Dado el siguiente `main`:

```
int main(int argc, char* argv[]) {
    int x = argc;
    return x + 33;
}
```

Si ejecutas ese código como `./prog`, que código de retorna da?

☒ 34

☐ 0

☐ 33

Enviar tarea