



Sintaxis y Semántica de los Lenguajes

Comparación GCC vs Clang



vs



Docente:

Esp. Ing. Jose Maria Sola

Alumno: Roberto Cuevas

Índice

1. Introducción
2. ¿Qué es GCC?
3. ¿Qué es Clang?
4. Arquitectura de GCC
5. Arquitectura de Clang
6. Diferencias en la arquitectura y el diseño
7. Diferencias funcionales
8. Comparación práctica mediante ejemplos en C
 - Ejemplo 1 - Comprobación de uso de variables no inicializadas
 - Ejemplo 2 - Comportamiento ante extensiones de GNU
 - Ejemplo 3 - Diferencia en tamaño del ejecutable generado
 - Ejemplo 4 - Detección de posibles errores lógicos
 - Ejemplo 5 - Compatibilidad del código
9. Conclusión General
10. Bibliografía

Comparación entre los compiladores GCC y Clang: arquitectura, diseño y diferencias prácticas

1. Introducción

En el desarrollo de software en lenguajes como C y C++, los compiladores desempeñan un rol fundamental al traducir el código fuente en binarios ejecutables eficientes. Entre las herramientas más utilizadas a nivel global se encuentran **GCC (GNU Compiler Collection)** y **Clang**, ambos ampliamente adoptados en sistemas basados en Unix y entornos de desarrollo modernos. Si bien ambos cumplen funciones similares, presentan diferencias significativas en su diseño, arquitectura interna, rendimiento, portabilidad y filosofía de desarrollo.

Esta monografía tiene como objetivo analizar y comparar las características estructurales de GCC y Clang, resaltando sus diferencias arquitectónicas, su impacto en el proceso de compilación y comportamiento práctico frente a casos concretos en el lenguaje C. Asimismo, se incluyen ejemplos que ilustran la forma en que cada compilador maneja determinadas construcciones del lenguaje, facilitando una evaluación crítica de sus ventajas y limitaciones.

2. Que es GCC

GCC (GNU Compiler Collection) es un conjunto de compiladores creado originalmente por el proyecto GNU, diseñado para compilar código en varios lenguajes de programación. Es ampliamente reconocido como una herramienta fundamental del software libre.

Según su documentación oficial, GCC ofrece las siguientes características principales:

- Es multilenguaje, dado que soporta C, C++, Objective-C, Fortran, Ada, Go entre otros.
- Es Multiplataforma, funciona con una gran variedad de arquitecturas y SO.
- Es de código abierto, bajo licencia GLP, lo que fomenta su mejora y distribución libre.
- Madurez y robustez, dado que esta en desarrollo desde 1987, y es utilizado en proyectos como Linux, GNOME y KDE

De acuerdo con su misión, GCC persigue los siguientes objetivos:

“Desarrollar y mantener una infraestructura de compilación libre, de alta calidad y compatible con estándares.”

En esencia, GCC busca ser un compilador confiable, portable y conforme a los estándares internacionales, proporcionando herramientas tanto para desarrolladores individuales como para grandes sistemas operativos.

3. Que es Clang

Clang es un compilador de C, C++ y Objective-C diseñado como parte del proyecto LLVM. Nació como una alternativa moderna a GCC, con el objetivo de ofrecer una arquitectura más modular y enfocada en el rendimiento, la claridad del código y la facilidad de integración con otras herramientas. Fue creado inicialmente por Apple entre 2007 y mantenido por la Fundación LLVM y su comunidad activa.

Según su documentación oficial, Clang ofrece las siguientes características principales:

- Basado en LLVM: Usa LLVM como backend, lo que permite análisis, optimización y generación de código en múltiples etapas.
- Errores más legibles: Proporciona mensajes de error y advertencias más claros que GCC.
- Compilación más rápida: Diseñado para ser eficiente en términos de tiempo y recursos.
- Arquitectura modular: Separa el frontend (análisis y parsing) del backend (generación de código), permitiendo su uso como librería.
- Mejor integración con herramientas modernas: Facilita su uso en IDEs y analizadores estáticos.

De acuerdo con su diseño

“Clang está diseñado para proporcionar una API completa que permita usar el compilador como una librería dentro de otras aplicaciones.”

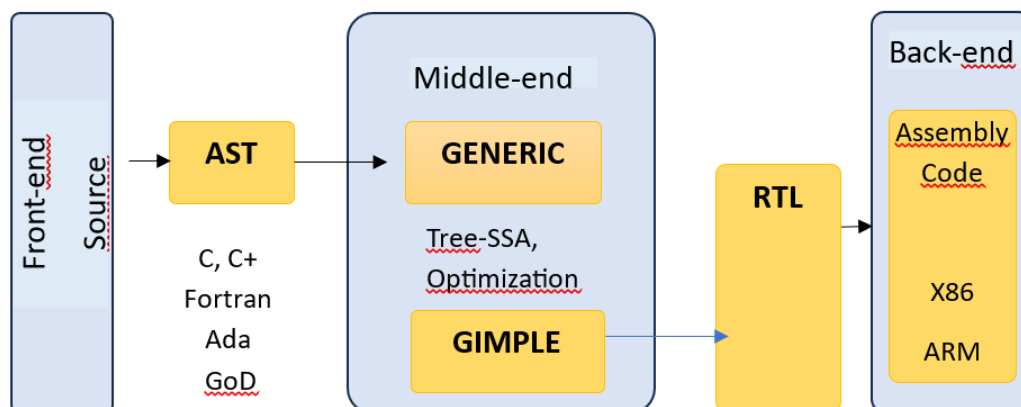
Esto significa que **Clang no solo es un compilador, sino también una plataforma de herramientas** que pueden analizar, transformar y compilar código fuente de manera muy flexible.

4. Arquitectura de GCC

La arquitectura de GCC está basada en un diseño monolítico, donde cada compilador para un lenguaje específico está estrechamente integrado dentro del sistema general. El proceso de compilación en GCC se divide en múltiples fases que incluyen: preprocesamiento, análisis léxico y sintáctico, generación de un árbol intermedio (GIMPLE), optimización y generación de código ensamblador.

GCC emplea un modelo de representación intermedia conocido como GIMPLE, una forma simplificada de árbol de sintaxis abstracta que facilita la aplicación de múltiples etapas de optimización. Luego, este árbol es transformado en RTL (Register Transfer Language), que representa operaciones de bajo nivel específicas para la arquitectura destino. Finalmente, el backend de GCC convierte el RTL en código ensamblador. Una de las principales características de GCC es su amplia compatibilidad con diversas arquitecturas de hardware, lo que ha sido posible gracias a su diseño adaptable y su soporte para múltiples backends.

Basicamente GCC adopta un diseño tradicional y monolítico con varias capas bien definidas:



Frontend: Es la etapa inicial de compilador, donde se toma el código fuente (C, C++, Fortran, Ada, etc.) y se convierte a una estructura intermedia, llamado AST (Abstract sintáctic tree) que representa la gramática del código fuente. Es una forma jerárquica que permite entender la lógica del programa sin preocuparse por los detalles sintácticos exactos. luego convierte el AST en la representación GENERIC, común a todos los lenguajes

Middle-end: transforma GENERIC a GIMPLE, una representación de tres direcciones (tres operandos por instrucción) que simplifica expresiones complejas y facilita optimizaciones independientes del lenguaje y la arquitectura usa el paso Tree-SSA para optimizaciones avanzadas como propagación de constantes, eliminación de redundancias, bucles, inlining, etc.

Back-end: convierte GIMPLE a RTL (Register Transfer Language), una RI de bajo nivel cercana al ensamblador, y finalmente produce código ensamblador específico usando patrones por arquitectura (usando macros de descripción de máquina)

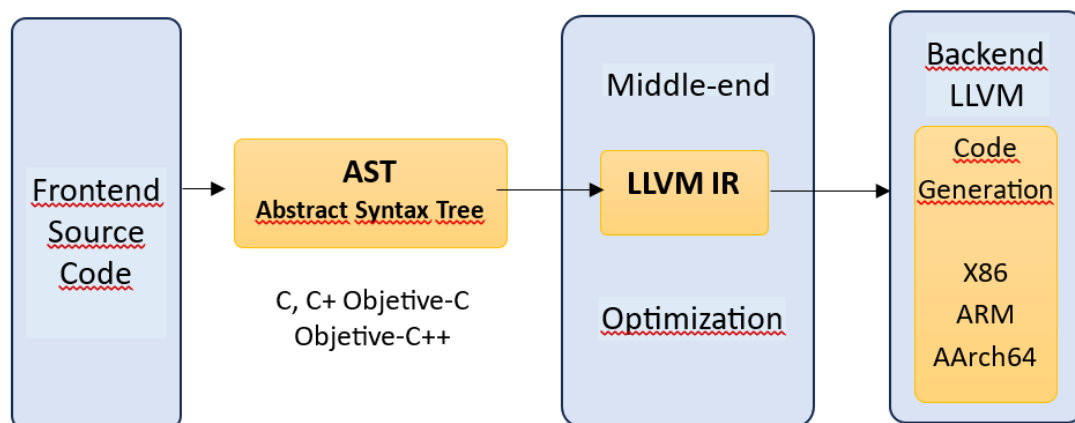
GCC tiene soporte para **Link-Time Optimization (LTO)**, que permite atravesar límites de archivo para hacer optimizaciones a nivel de programa completo. Además, su diseño permite extensiones mediante plugins, interfaces para nuevos frontends y soporte de nuevas arquitecturas.

5. Arquitectura de CLANG

Clang es un compilador moderno diseñado para ser una interfaz de frontend del proyecto LLVM. Su arquitectura se distingue por la modularidad, claridad en el diseño y una fuerte orientación hacia el desarrollo de herramientas. Mientras que GCC está construido como un sistema monolítico, Clang separa rigurosamente las responsabilidades del compilador en etapas bien definidas, facilitando la reutilización de sus componentes.

Estructura General

La arquitectura de Clang se fundamenta en un diseño modular y puede dividirse en cuatro bloques principales:



Frontend

Se encarga de analizar el código fuente en C, C++, Objective-C y Objective-C++. Su funcionamiento incluye

- Tokenización y análisis léxico
- Análisis sintáctico (parser)
- Chequeo semántico
- Generación del AST

Clang se caracteriza por producir mensajes de error y advertencias detallados y legibles, ayudando a los desarrolladores a comprender rápidamente los problemas en el código.

AST (Abstract Syntax Tree)

El AST generado por Clang es rico y detallado, diseñado no solo para compilación, sino también para habilitar herramientas de análisis estático, refactorización y desarrollo asistido. Este árbol se mantiene accesible como una estructura de datos reutilizable, lo que lo convierte en una base poderosa para herramientas como:

- clang-tidy (análisis estático: analizar archivos fuente sin ejecutarlos)
- clang-format (formato automático)
- libclang (API para editores y asistentes de código)

LLVM IR (Intermediate Representation)

Luego del análisis del AST, Clang traduce el código fuente al LLVM IR, un lenguaje intermedio de tres niveles (High, Mid y Low Level IR). Este IR es independiente del lenguaje fuente y del hardware, lo que permite aplicar un conjunto común de optimizaciones agresivas sobre el código.

Ventajas del uso de LLVM IR:

- Optimización universal del código
- Posibilidad de generar binarios para múltiples arquitecturas sin modificar el frontend
- Reutilización del mismo backend para diferentes lenguajes (no solo C/C++)

LLVM Backend

El LLVM IR se traduce finalmente en código máquina mediante el backend de LLVM. Esta etapa incluye:

- Selección de instrucciones específicas de la arquitectura destino (x86, ARM, RISC-V, etc.)
- Asignación de registros
- Generación de código ensamblador
- Enlazado con bibliotecas y generación de ejecutables

6. Diferencias en la arquitectura y el diseño

El diseño interno de un compilador influye de forma directa en su facilidad de mantenimiento, capacidad de expansión, nivel de optimización y en la experiencia del desarrollador. Aunque tanto **GCC** (GNU Compiler Collection) como **Clang** (como frontend de **LLVM**) compilan lenguajes como C y C++, sus enfoques arquitectónicos y objetivos de diseño son distintos. A continuación se detallan las diferencias más relevantes.

Modularidad: GCC implementa una arquitectura integrada y coherente, mientras que Clang/LLVM se compone de módulos reutilizables e independientes.

Representaciones intermedias: GCC emplea una cadena de transformaciones que incluye GENERIC, GIMPLE y RTL. Clang, en cambio, traduce directamente al LLVM IR, reduciendo la complejidad y estandarizando el proceso de optimización.

Diagnóstico de errores: Clang se destaca por su sistema de diagnóstico avanzado, con mensajes de error más descriptivos y precisos, lo que mejora la experiencia del desarrollador.

Facilidad de integración: Clang, gracias a su arquitectura basada en bibliotecas reutilizables, se integra fácilmente en herramientas como IDEs y analizadores estáticos.

Extensibilidad y mantenibilidad: LLVM fue diseñado desde sus orígenes orientado a facilitar la incorporación de nuevos lenguajes, optimizaciones y otras arquitecturas, mientras que GCC, requiere de mas recursos estructurales para incorporar cambios significativos.

Interoperabilidad: La arquitectura de LLVM permite desarrollar herramientas adicionales como analizadores de código, generadores de documentación y compiladores cruzados de forma más sencilla que con GCC.

Arquitectura y diseño	GCC	Clang / LLVM
Modularidad	Arquitectura integrada: componentes acoplados.	Arquitectura modular: componentes reutilizables.
Representación intermedia (IR)	Usa IR internas (GENERIC, GIMPLE, RTL)	Usa LLVM IR: estándar documentada y compartida
Mensajes de error	Diagnóstico tradicional: mensajes técnicos y menos descriptivos.	Diagnóstico avanzado: mensajes claros, precisos y con sugerencias.
Facilidad de integración	Difícil de integrar debido a diseño monolítico.	Fácil de integrar en IDEs y herramientas - API pública.
Extensibilidad y mantenimiento	Requiere mas recursos para modificar o extender; código más complejo.	Diseñado para extensible: fácil de mantener y expandir a nuevos lenguajes.
Interoperabilidad	Limitada: difícil construir herramientas adicionales externas.	Alta: permite desarrollo de analizadores, refactorizadores, compiladores cruzados.
Separación Frontend/Backend	Frontend y backend unidos; arquitectura menos flexible.	Frontend (Clang) separado del backend (LLVM); adaptable y reutilizable.
Uso como librería (API)	No pensado para ser usado como API externa.	Disponible como librería: se puede usar desde otras aplicaciones fácilmente.

7. Comparación funcional entre GCC y Clang

Los compiladores GCC y Clang implementan las funciones fundamentales de un compilador moderno, como el análisis del código fuente, la generación de representaciones intermedias, la optimización y la producción de código ejecutable. Sin embargo, difieren en la forma en que están estructurados internamente y en cómo estas funciones se organizan y ejecutan dentro de sus arquitecturas.

Análisis léxico y sintáctico

Ambos compiladores cuentan con analizadores capaces de procesar lenguajes como C y C++. En Clang, esta etapa se encuentra dentro de un frontend diseñado específicamente para producir mensajes de diagnóstico claros, precisos y fáciles de comprender, según se destaca en su documentación oficial. GCC también cuenta con analizadores robustos y maduros, producto de décadas de evolución, orientados a garantizar compatibilidad con estándares y múltiples lenguajes.

Análisis semántico y diagnóstico de errores

En Clang, el análisis semántico se realiza sobre una estructura de árbol sintáctico abstracto (AST) construida para facilitar la identificación de errores en etapas tempranas. Según clang.llvm.org, el sistema de diagnósticos de Clang está diseñado desde el inicio para ofrecer sugerencias contextuales y mensajes localizados. GCC también implementa un sistema de diagnóstico eficiente, aunque la documentación oficial no enfatiza detalles sobre la presentación de advertencias o errores.

Generación de código intermedio

GCC transforma el código fuente en una serie de representaciones intermedias: primero GENERIC, luego GIMPLE, y finalmente RTL (Register Transfer Language), que permite aplicar optimizaciones progresivas a distintos niveles. Por su parte, Clang transforma el código fuente en un AST que se convierte directamente en LLVM IR (Intermediate Representation), una forma intermedia modular y reutilizable dentro del ecosistema LLVM. Esta diferencia estructural representa uno de los contrastes fundamentales entre ambos compiladores.

Optimización y backend

Ambos compiladores aplican múltiples estrategias de optimización sobre su representación intermedia. En GCC, estas optimizaciones se aplican mayormente sobre GIMPLE y luego sobre RTL, antes de generar el código máquina final. En Clang, el LLVM IR permite aplicar optimizaciones en una infraestructura diseñada para ser extensible y desacoplada del lenguaje fuente. Finalmente, ambos compiladores generan código objeto compatible con múltiples arquitecturas.

Diseño modular y reutilización

Mientras que GCC sigue un diseño monolítico con integración estrecha entre etapas, Clang y LLVM adoptan un diseño modular orientado a la reutilización de componentes. Esto permite que herramientas de análisis, compilación y depuración puedan compartir elementos como el frontend o el backend.

Conclusión

GCC y Clang implementan todas las etapas de un compilador, pero los enfoques difieren según su diseño interno y objetivos de arquitectura. GCC se caracteriza por una arquitectura monolítica y probada, con múltiples representaciones intermedias que permiten una optimización eficaz. Clang, en cambio, se construye sobre una arquitectura modular centrada en la reutilización, claridad diagnóstica y extensibilidad, lo cual lo hace una herramienta especialmente adecuada para entornos de desarrollo modernos y herramientas externas.

Aunque ambos compiladores cumplen eficientemente con sus funciones, Clang destaca en aspectos relacionados con la experiencia del desarrollador y la integración de herramientas, mientras que GCC sobresale por su madurez, optimizaciones específicas y compatibilidad amplia. Esta diversidad de enfoques demuestra que cada herramienta responde a necesidades distintas, y su elección dependerá del contexto de uso, los objetivos del proyecto y los requerimientos técnicos.

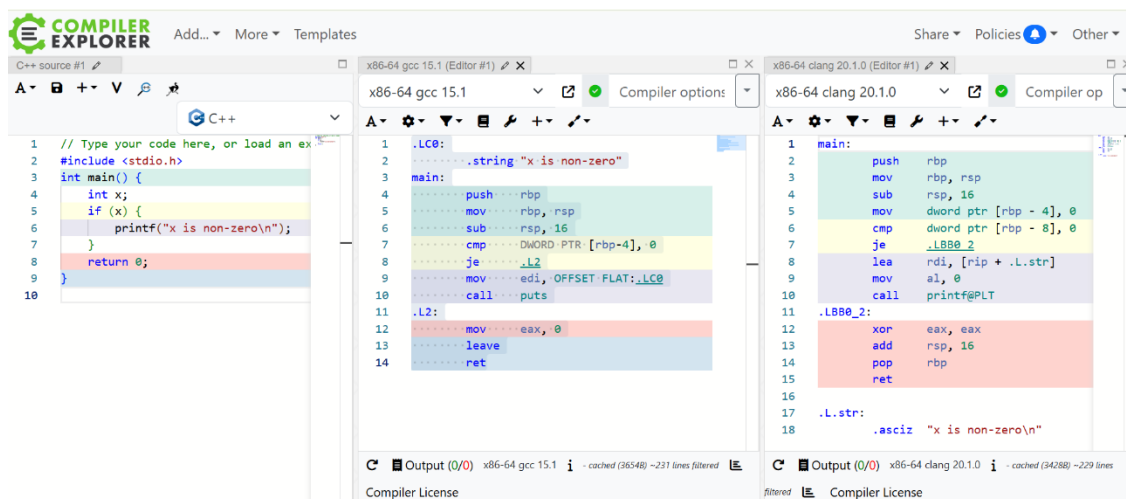
8. Comparativa práctica mediante ejemplos en C

Para ilustrar las diferencias prácticas entre GCC y Clang, se presentan a continuación algunos ejemplos simples en lenguaje C, analizando la generación de código en ambos compiladores.

Ejemplo 1 – Comprobación de uso de variables no inicializadas

```
#include <stdio.h>
int main() {
    int x;
    if (x) {
        printf("x is non-zero\n");
    }
    return 0;
}
```

Compilación online con gcc 15.1 y clang 20.1.0 de godbolt.org



<u>GCC</u>	<u>CLANG</u>
Usa puts para imprimir la cadena	Usa printf, no puts
call puts	Call printf@PLT
No inicializa x, lo compara directamente	Inicializa x en 0
Cmp DWORD PTR [rbp-4], 0	mov dword ptr [rbp - 4], 0
Carga el string desde una etiqueta	La cadena se guarda en
.LC0	.L.str con \n incluido

Conclusiones:

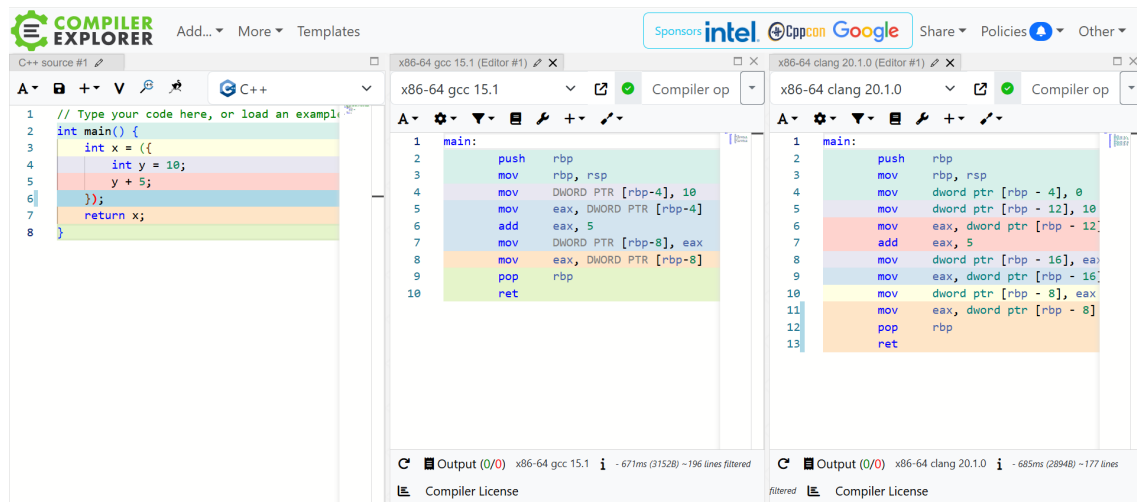
Clang podría protegernos más inicializando por defecto, GCC asume que el programador sabe lo que hace (más fiel al standard C puro).

Tienen funciones de salida distintas, como el mensaje no tiene formato, GCC lo optimiza usando puts. Clang no lo hace y usa printf, que consume más recursos.

Ejemplo 2 – Comportamiento ante extensiones de GNU (GNU Not Unix)

```
int main() {
    int x = ({
        int y = 10;
        y + 5;
    });
    return x;
}
```

Compilación online con compiler explorer gcc 15.1 y clang 20.1.0 de godbolt.org



The screenshot displays the Compiler Explorer interface with the following components:

- Source Code:** C++ code defining `int x = ({ int y = 10; y + 5; });` and `return x;`.
- Compiler Selection:** GCC 15.1 and Clang 20.1.0 are selected for x86-64.
- Assembly Output:**
 - GCC 15.1:** Shows assembly instructions for `main`, including `push rbp`, `mov rbp, rsp`, `mov DWORD PTR [rbp-4], 10`, `mov eax, DWORD PTR [rbp-4]`, `add eax, 5`, `mov DWORD PTR [rbp-8], eax`, `mov eax, DWORD PTR [rbp-8]`, `pop rbp`, and `ret`.
 - Clang 20.1.0:** Shows assembly instructions for `main`, including `push rbp`, `mov rbp, rsp`, `mov dword ptr [rbp-4], 0`, `mov dword ptr [rbp-12], 10`, `mov eax, dword ptr [rbp-12]`, `add eax, 5`, `mov dword ptr [rbp-16], eax`, `mov eax, dword ptr [rbp-16]`, `mov dword ptr [rbp-8], eax`, `mov eax, dword ptr [rbp-8]`, `pop rbp`, and `ret`.
- Output:** Both compilers show `Output (0/0)` with no errors or warnings.

Análisis técnico y conclusión

Estas instrucciones son conocidas como **statement expression** que suma una o mas sentencias dentro de `{ ... }` para que se comporte como una expresión y retorna un valor.

GCC soporta nativamente la construcción como parte de su conjunto de extensiones. El código se compila y el valor de 15 se asigna correctamente a `x`.

CLANG, también acepta esta extensión por compatibilidad con GCC. El código generado es un poco distinto, pero hace las mismas funciones: declara `y`, evalúa `y + 5`, y lo retorna como valor de `x`.

Este caso demuestra cómo un compilador como CLANG, pese a no pertenecer al serie de sistemas de GNU, implementa extensiones de GCC para garantizar la compatibilidad.

Ejemplo 3 – Diferencia en tamaño del ejecutable generado

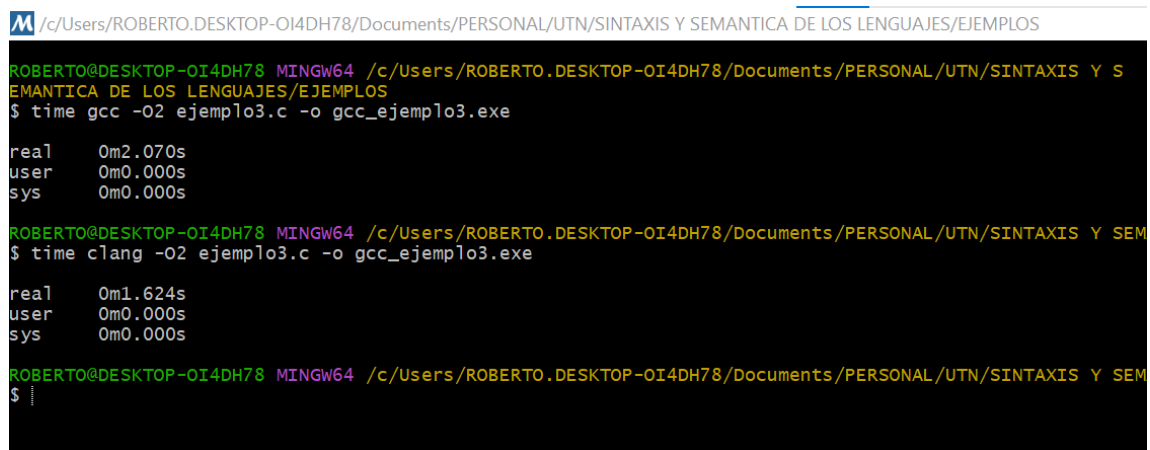
```
#include <stdio.h>
#include <string.h>

int suma_ascii(const char* str) {
    int total = 0;
    for (int i = 0; i < strlen(str); i++) {
        total += str[i];
    }
    return total;
}

int main() {
    char nombre[] = "CLANG vs GCC";
    printf("Suma ASCII: %d\n", suma_ascii(nombre));
    return 0;
}
```

Compilación con MSYS2 MinGW 64-bit gcc version 15.1.0 y clang version 20.1.8

Compilación midiendo la velocidad y optimización general



```
/c/Users/ROBERTO.DESKTOP-OI4DH78/Documents/PERSONAL/UTN/SINTAXIS Y SEMANTICA DE LOS LENGUAJES/EJEMPLOS
ROBERTO@DESKTOP-OI4DH78 MINGW64 /c/Users/ROBERTO.DESKTOP-OI4DH78/Documents/PERSONAL/UTN/SINTAXIS Y SEMANTICA DE LOS LENGUAJES/EJEMPLOS
$ time gcc -O2 ejemplo3.c -o gcc_ejemplo3.exe

real    0m2.070s
user    0m0.000s
sys     0m0.000s

ROBERTO@DESKTOP-OI4DH78 MINGW64 /c/Users/ROBERTO.DESKTOP-OI4DH78/Documents/PERSONAL/UTN/SINTAXIS Y SEMANTICA DE LOS LENGUAJES/EJEMPLOS
$ time clang -O2 ejemplo3.c -o gcc_ejemplo3.exe

real    0m1.624s
user    0m0.000s
sys     0m0.000s

ROBERTO@DESKTOP-OI4DH78 MINGW64 /c/Users/ROBERTO.DESKTOP-OI4DH78/Documents/PERSONAL/UTN/SINTAXIS Y SEMANTICA DE LOS LENGUAJES/EJEMPLOS
$
```

Confirmación del tamaño de los ejecutables generados



```
ROBERTO@DESKTOP-OI4DH78 MINGW64 /c/Users/ROBERTO.DESKTOP-OI4DH78/Documents/PERSONAL/UTN/SINTAXIS Y SEMANTICA DE LOS LENGUAJES/EJEMPLOS
$ size gcc_ejemplo3.exe
size clang_ejemplo3.exe
  text  data  bss   dec    hex filename
41204   272   2944  44420  ad84 gcc_ejemplo3.exe
  text  data  bss   dec    hex filename
39760   272   2944  42976  a7e0 clang_ejemplo3.exe
```

Análisis técnico y conclusión

Sección text (código ejecutable) Clang genera código más compacto (3.25% menor)

GCC: 41204 bytes

Clang: 39760 bytes

Sección data y bss (datos inicializados y sin inicializar)

Son iguales entre ambos 272 y 2944 respectivamente

Tamaño total (dec)

GCC: 44420 bytes

Clang: 42976 bytes

Podemos ver que el ejecutable generado por **Clang** fue aproximadamente un 3,25% más pequeño que el generado por **GCC**. Esta diferencia proviene fundamentalmente del tamaño de la sección .text, donde **Clang** generó un código objeto más compacto.

Esta diferencia parece no ser significativa, pero en entornos donde el espacio de almacenamiento y memoria es crítico, una reducción de tamaño como la que proporciona **Clang** podría ser ventajosa.

Clang tiende a generar ejecutables más compactos que GCC en igualdad de condiciones de optimización, lo que puede representar una ventaja concreta en ciertos escenarios.

Además del tamaño del archivo, el tiempo real de compilación de GCC fue de 0m2.070s, 2 minutos 7 segundos, cuando el tiempo de compilación de CLANG fue de 0m1.624s. 1 minuto 624s. demostrando que es significativamente menor.

Ejemplo 4 – Detección de posibles errores lógicos

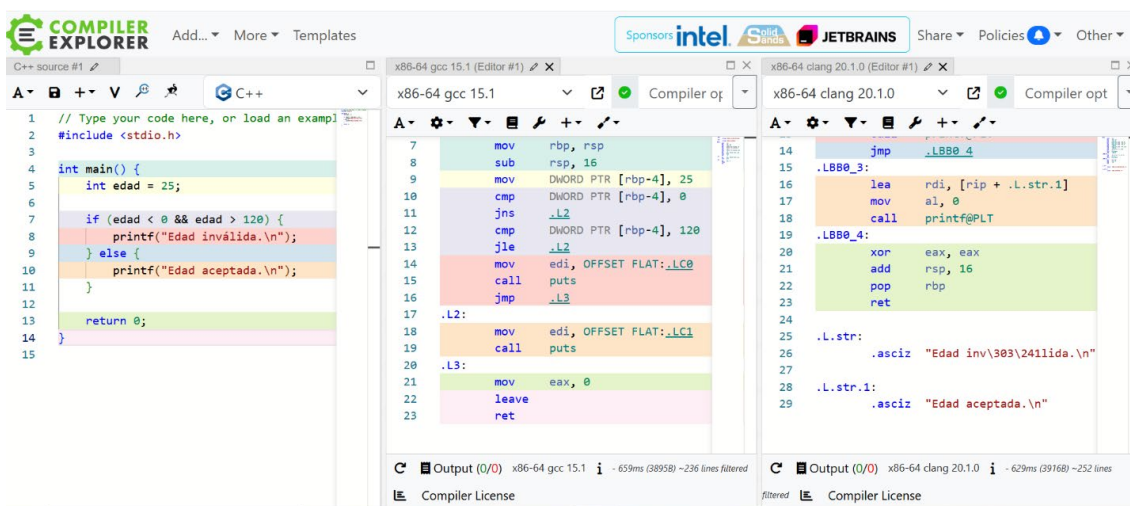
```
#include <stdio.h>

int main() {
    int edad = 25;

    if (edad < 0 && edad > 120) {
        printf("Edad inválida.\n");
    } else {
        printf("Edad aceptada.\n");
    }
    return 0;
}
```

En este ejemplo La condición `edad < 0 && edad > 120` nunca será verdadera, dado que la variable no puede ser simultáneamente menor que 0 y mayor que 120. Esto constituye un error lógico típico, difícil de detectar si no se revisa manualmente o se usan herramientas de análisis estático.

Compilación online con compiler explorer gcc 15.1 y clang 20.1.0 de godbolt.org



The screenshot displays the Compiler Explorer interface with the following C++ code in the editor:

```
1 // Type your code here, or load an example
2 #include <stdio.h>
3
4 int main() {
5     int edad = 25;
6
7     if (edad < 0 && edad > 120) {
8         printf("Edad inválida.\n");
9     } else {
10        printf("Edad aceptada.\n");
11    }
12
13    return 0;
14 }
```

The interface shows two compilation results side-by-side:

- Left Panel (GCC 15.1):** Shows assembly code for x86-64 gcc 15.1. The assembly includes instructions for setting up the stack, comparing the value 0 with the variable `edad`, and then comparing `edad` with 120. It then branches based on the result of the second comparison. The output shows the program running successfully without any warnings or errors.
- Right Panel (Clang 20.1.0):** Shows assembly code for x86-64 clang 20.1.0. The assembly is similar to the GCC version but includes a warning message: `Warning: logical AND of contradictory conditions [-Wlogical-and-contradictory-conditions]`. The output shows the program running successfully with the warning.

Análisis técnico y conclusión:

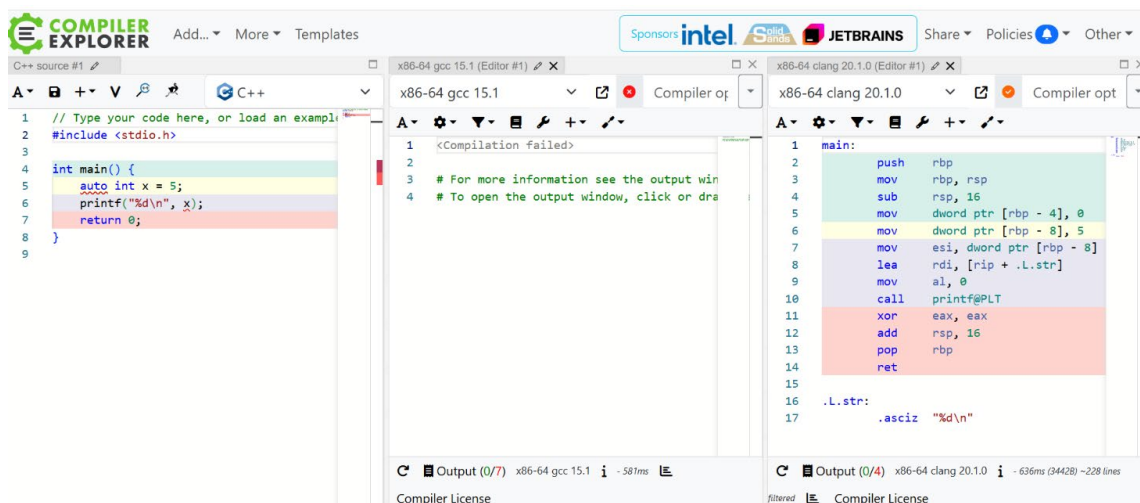
En el ejercicio se comparó el comportamiento de Clang y GCC ante una condición lógica contradictoria. Clang mostró una advertencia señalando que la comparación lógica nunca puede ser verdadera, ayuda al programador a detectar un posible error de lógica. Al contrario, GCC no emitió ninguna advertencia con las opciones de compilación estándar, lo que podría llevar a no detectar este tipo de falla si no se realiza un análisis detallado. Esta diferencia evidencia la fortaleza de Clang en la detección de errores lógicos sutiles.

Ejemplo 5 – Compatibilidad código

```
#include <stdio.h>
```

```
int main() {
    auto int x = 5;
    printf("%d\n", x);
    return 0;
}
```

Compilación online con compiler explorer gcc versión 15.1 y clang versión 20.1.0



Análisis técnico y conclusión

GCC 15.1 no permite compilar este código y lanza un error de compilación.
error: expected '=', ',', ';, 'asm' or '__attribute__' before 'x'

Clang 20.1.0, compila correctamente por estar configurado para compilar como C puro y acepta `auto int` como válido, y interpretándolo como una variable de tipo entero con almacenamiento automático.

Este ejemplo muestra la diferencia en la interpretación del estándar o modo de compilación que puede generar resultados distintos. El código fuente es válido en diferentes versiones de C, pero la configuración del compilador y la compatibilidad puede hacer que el mismo código compile correctamente en Clang, y falle en GCC, al considerar inválido el uso de `auto int`.

Resumen de los resultados prácticos

Tema del ejemplo	GCC	Clang
Variables no inicializadas	No advierte salvo con flags específicos	Advierte por defecto con -Wall
Extensiones GNU	Las acepta como parte integral del compilador	Las soporta por compatibilidad
Tamaño del ejecutable	Ejecutable más grande en general	Ejecutable más compacto
Errores lógicos en condiciones	No detecta condiciones imposibles por defecto	Advierte sobre lógica inválida (&&)
Compatibilidad de sintaxis	Rechaza con error en configuración actual	Acepta y compila normalmente

9. Conclusión General

A lo largo de esta monografía se han intentado mostrar las principales diferencias entre los compiladores GCC y Clang, desde su origen, desarrollo y estructura, hasta su comportamiento frente a diversos casos de programación en lenguaje C. A través de ejemplos prácticos se evidenciaron aspectos fundamentales de un compilador.

En términos generales, Clang se destacó por ofrecer diagnósticos más precisos, advertencias detalladas y una tendencia a generar ejecutables más eficientes. GCC, por su parte, mostró una sólida integración con el ecosistema GNU y una mayor tolerancia a construcciones de versiones anteriores del lenguaje.

Estos resultados permiten concluir que la elección de un compilador no debe basarse únicamente en el rendimiento, sino también en que contexto se va a realizar el proyecto, la portabilidad deseada, el estándar de lenguaje requerido y el tipo de control que se busca durante el desarrollo. Tanto GCC como Clang son herramientas poderosas y activamente mantenidas, y conocer sus diferencias puede mejorar significativamente la calidad del código y la eficiencia del proceso de desarrollo.

10. Bibliografía

- GCC Mission Statement. (n.d.). Descargado de <https://gcc.gnu.org/gccmission.html>
- GCC Internals Manual. Free Software Foundation. (2024). Top (GNU Compiler Collection (GCC) Internals). <https://gcc.gnu.org/onlinedocs/gccint/>
- Using the GNU Compiler Collection. 2023 Free Software Foundation, Inc.
<http://www.gnupress.org>
- Clang 20 Documentation. LLVM Project. (2024). Clang Compiler User's Manual.
<https://clang.llvm.org/docs/>
- An Overview of clang. Stulova-Haastregt (2019). Arm, Cambridge, UK
- ISO/IEC 9899:2018 (C17 Standard). International Organization for Standardization. (2018). Programming languages — C. ISO.
- Compiler Explorer (godbolt.org). Godbolt, M. (2024). Compiler Explorer.
<https://godbolt.org/>
- LLVM Blog & Release Notes. LLVM Foundation. (2024). <https://llvm.org/blog/>
- El Lenguaje de Programación C – Kernighan, Ritchie
- Ejemplo ilustrativo basado en el comportamiento documentado en la sección de diagnósticos del compilador Clang (<https://clang.llvm.org/features.html>)
<https://www.gnu.org/prep/standards/>
- Material de clases de la asignatura SSL – Ing. Sola - CompilersEditorsIdes.pdf
- Material de clases de la asignatura SSL – Ing. Sola – Git 101