



UNIVERSIDAD NACIONAL AUTÓNOMA DE MÉXICO

Facultad de Ciencias. Compiladores

Práctica 0

Reporte de Práctica

Autor:

Rosas Marín Jesús Martín - **318291015**

Profesores:

Ariel Adara Mercado Martínez
Carlos Gerardo Acosta Hernández
Erick Bernal Márquez
Yessica Janeth Pablo Martínez
Kevin Isaac Alcantara Estrada

12 de febrero de 2025

Resultados

1. El siguiente script en c se guardo en un archivo llamado **programa.c** :

```
C programa.c > ...
1  #include <stdio.h>
2  #include <stdlib.h>
3  //# define PI 3.1415926535897
4
5  # ifdef PI
6  # define area(r)(PI * r * r)
7  # else
8  # define area(r)(3.1416 * r * r)
9  # endif
10
11 |
12 /**
13  * Compiladores 2025-1
14  *
15  */
16 int main( void ) {
17     printf ("Hola Mundo !\n"); //Función para imprimir hola mundo
18     float mi_area = area(3); //soy un comentario... hasta donde llegare ?
19     printf ("Resultado : %f\n", mi_area);
20     return 0;
21 }
```

2. Use el siguiente comando: **cpp programa.c programa.i**

Revise el contenido de programa.i y conteste lo siguiente:

- a) ¿Qué ocurre cuando se invoca el comando cpp con esos argumentos?

Lo primero que podemos notar es que se crea el archivo **programa.i**.

Ahora, en el archivo generado, podemos ver que todas las directivas de preprocesamiento como `#include <stdio.h>` y `#include <stdlib.h>` son reemplazadas por el contenido completo de esos archivos de cabecera, esto agrega muchas mas líneas de código.

También veamos que como PI no está definido, el preprocesador usa la parte `#else` de la directiva `#ifdef`. La macro `area(r)` se va a expandir a `(3.1416 * r * r)`.

```
# 16 "programa.c"
int main( void ) {
    printf ("Hola Mundo !\n");
    float mi_area = (3.1416 * 3 * 3);
    printf ("Resultado : %f\n", mi_area);
    return 0;
}
```

En el código, `area(3)` se reemplaza por `(3.1416 * 3 * 3)`.

En general, el preprocesador (`cpp`) toma el archivo fuente `programa.c` y aplica todas las directivas de preprocesamiento.

- b) ¿Qué similitudes encuentra entre los archivos **programa.c** y **programa.i**?

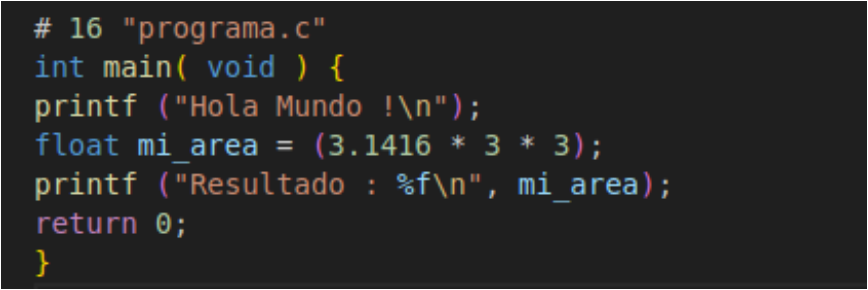
Podemos notar que en primera, ambos archivos contienen la estructura básica del programa, como la función `main` y las llamadas a `printf`.

Como mencionamos en el punto anterior, podemos notar que las macros definidas en `programa.c` (como `area(r)`) están presentes en `programa.i`, pero ya han sido expandidas. Por ejemplo, `area(3)` se convierte en `3.1416 * 3 * 3`.

En general podemos decir que ambos archivos contienen la misma lógica y estructura del programa y que las funciones y macros se conservan solo que las macros ya han sido expandidas en `programa.i`.

- c) ¿Qué pasa con las macros y los comentarios del código fuente original en **programa.i**?

Primero, veamos de nuevo el `main` en **programa.i**:



```
# 16 "programa.c"
int main( void ) {
printf ("Hola Mundo !\n");
float mi_area = (3.1416 * 3 * 3);
printf ("Resultado : %f\n", mi_area);
return 0;
}
```

En primera instancia podemos ver que los comentarios del código fuente original son eliminados por el preprocesador. Esto incluye tanto los comentarios de una línea como los comentarios de varias líneas.

Ahora veamos que las macros ya no aparecen como definiciones en **programa.i** sino que el preprocesador ha reemplazado todas las instancias de la macro con su valor expandido.

- d) Compare el contenido de **programa.i** con el de **stdio.h** e indique de forma general las similitudes entre ambos archivos.

Podemos notar que ambos archivos contienen declaraciones de funciones de la biblioteca estándar de C como `printf` y que ambos pueden contener definiciones de tipos y macros.

Pero también podemos ver diferencias como que en el código de **programa.i** viene el código del programa (`main` y las llamadas a funciones), mientras que `stdio.h` solo contiene declaraciones y definiciones.

Pero en conclusión, la estructura general de ambos archivos es similar, ya que `programa.i` incluye el contenido de `stdio.h`.

- e) ¿A qué etapa corresponde este proceso?

Este proceso corresponde a la etapa de preprocesamiento ya que el preprocesador (invocado con el comando `cpp`) realiza las tareas de:

-
- 1) Expansión de macros (reemplaza las macros por su definición).
 - 2) Inclusión de archivos de cabecera (include).
 - 3) Eliminación de comentarios.
 - 4) Procesamiento de directivas (ifdef, if, else, endif, etc.).

3. Ejecute la siguiente instrucción: **gcc -Wall -S programa.i**

a) ¿Para qué sirve la opción -Wall?

Si investigamos un poco sobre esta bandera encontraremos que su nombre proviene de "Warnings All" y lo que hace es habilitar un conjunto de advertencias de compilación que ayudan a detectar posibles errores o problemas en el código.

También el usar -Wall, el compilador va a alertar sobre prácticas que, aunque no necesariamente causen errores de compilación, podrían llevar a comportamientos inesperados en tiempo de ejecución.

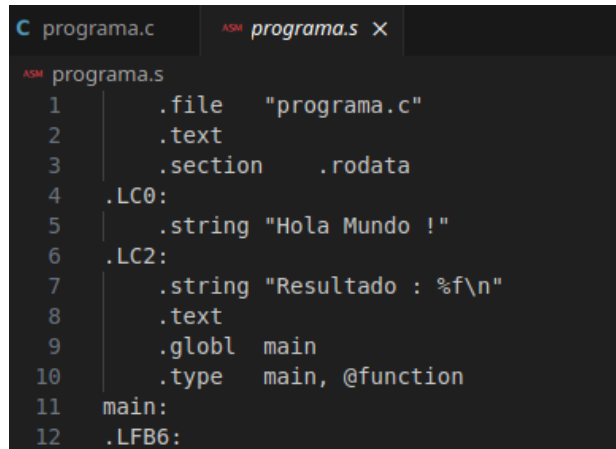
b) ¿Qué le indica a gcc la opción -S?

La opción -S en el comando gcc le indica al compilador que genere un archivo de código ensamblador (un archivo con extensión .s) a partir del archivo de entrada, en lugar de generar un archivo objeto o un ejecutable.

c) ¿Qué contiene el archivo de salida y cuál es su extensión?

Como mencionamos anteriormente, su extensión será .s, además de que el archivo de salida generado contiene código ensamblador.

Veamos como se ve:



```
programa.s
1  .file "programa.c"
2  .text
3  .section .rodata
4  .LC0:
5  .string "Hola Mundo !"
6  .LC2:
7  .string "Resultado : %f\n"
8  .text
9  .globl main
10 .type main, @function
11 main:
12 .LFB6:
```

Podemos notar que hay Directivas del ensamblador como:

- .section
- .globl

Tambien podemos notar que hay etiquetas y referencias como:

- .LC0: para referenciar direcciones de memoria o constantes.
- .LC2: apunta a una constante o cadena de texto almacenada en la sección

d) ¿A qué etapa corresponde este comando?

Veamos que esta etapa corresponde a la etapa de compilación ya que el compilador (invocado con `gcc`) toma el archivo preprocesado (`programa.i`) y lo traduce a código ensamblador. Y con la opción `-S` le indica a `gcc` que genere un archivo de código ensamblador `.s`

4. Ejecute la siguiente instrucción: **as programa.s -o programa.o**

a) Antes de revisarlo, indique cuál es su hipótesis sobre lo que debe contener el archivo con extensión .o.

Nuestra hipótesis es que el archivo .o resultante tendrá código máquina (binario) que es directamente ejecutable por el procesador, pero aún no es un programa completo.

Y las instrucciones de bajo nivel que vimos que habia (como mov, push, call, etc.) se traducen a sus equivalentes en código máquina.

Por lo que la hipotesis es que sera un archivo en codigo binario, donde estas instrucciones son la traducción de las instrucciones en ensamblador que estaban en el archivo programa.s.

b) Diga de forma general qué contiene el archivo programa.o y por qué se visualiza de esa manera.

Veamos primero el archivo:

```

1  ELF > @ @
2  0 0U000000H0 H000 0 E0f 000 ZM0fH ~0fH n0H0 H0N 0
3  010A GCC: (Ubuntu 11.4.0-1ubuntu1~22.04) 11.4.0 GNU 0
4  M 00

```

Por lo que podemos ver, contiene información en formato binario que representa el código máquina y los datos del programa, pero aún no es un ejecutable completo.

c) ¿Qué programa se invoca con as?

El comando `as` invoca el ensamblador, que es una herramienta que convierte código ensamblador (archivos `.s`) en código objeto (archivos `.o`).

d) ¿A qué etapa corresponde la llamada a este programa?

El ensamblador (as) convierte el código ensamblador (.s) en código objeto (.o).

Por lo que corresponde a la etapa de ensamblado.

5. Encuentre la ruta de los siguientes archivos en el equipo de trabajo:

- Scrt1.o
- crti.o
- crtbeginS.o
- crtendS.o
- crtn.o

Para encontrar la ruta de cada archivo utilice el comando **find** de la siguiente manera:

```
martillo@martillo:~/Desktop$ find / -iname "Scrt1.o"  
/usr/lib/x86_64-linux-gnu/Scrt1.o
```

Este comando se hizo de manera analoga con los 5 archivos, dandonos que todos estaban en la ruta:

/usr/lib/x86_64-linux-gnu/

6. Ejecute el siguiente comando, sustituyendo las rutas que encontró en el paso anterior:

`ld -o ejecutable -dynamic-linker /lib/ld-linux-x86-64.so.2 /usr/lib/crt1.o /usr/lib/crti.o programa.o -lc /usr/lib/crtn.o`

a) Describa el resultado obtenido al ejecutar el comando anterior.

```
martillo@martillo:~/Desktop$ ld -o ejecutable -dynamic-linker /usr/lib64/ld-linux-x86-64.so.2 /usr/lib/x86_64-linux-gnu/crt1.o /usr/lib/x86_64-linux-gnu/crti.o programa.o -lc /usr/lib/x86_64-linux-gnu/crtn.o  
martillo@martillo:~/Desktop$
```

Veamos que al ejecutar este comando, se genera nuestro ejecutable:

```
martillo@martillo:~/Desktop$ ls  
ejecutable Ejercicio programa.c programa.i programa.o programa.s  
martillo@martillo:~/Desktop$
```

7. Una vez que se enlazó el código máquina relocizable, podemos ejecutar el programa con la siguiente instrucción en la terminal: `./programa`

Veamos nuestro resultado:

```
martillo@martillo:~/Desktop$ ./ejecutable  
Hola Mundo !  
Resultado : 28.274401
```

8. Quite el comentario de la macro `#define PI` en el código fuente original y conteste lo siguiente:

```

C programa.c > ...
1  #include <stdio.h>
2  #include <stdlib.h>
3  # define PI 3.1415926535897
4
5  # ifdef PI
6  # define area(r)(PI * r * r)
7  # else
8  # define area(r)(3.1416 * r * r)
9  # endif
10
11
12  /**
13   * Compiladores 2025-1
14   *
15   */
16  int main( void ) {
17      printf ("Hola Mundo !\n"); //Función para imprimir hola mundo
18      float mi_area = area(3); //soy un comentario... hasta donde llegaré ?
19      printf ("Resultado : %f\n", mi_area);
20      return 0;
21  }

```

a) Genere nuevamente el archivo.i. De preferencia asigne un nuevo nombre.

De la misma manera que hicimos en el inciso 2, usaremos el comando **cpp programa.c programa.i**

b) ¿Cambia en algo la ejecución final?

Veamos que en este caso ya no entrara al else ya que si existe PI, por lo que se usara:

area(r)(PI * r * r)

Veamos ahora el resultado de la ejecución final:

```

martillo@martillo:~/Desktop$ ./ejecutable
Hola Mundo !
Resultado : 28.274334

```

Veamos que si cambia, el resultado al usar PI ahora es mas preciso.

9. Escribe un segundo programa en lenguaje C en el que agregue 4 directivas del preprocesador de C (cpp). Las directivas elegidas deben jugar algún papel en el significado del programa, ser distintas entre sí y diferentes de las utilizadas en el primer programa (aunque no están prohibidas si las requieren).

Veamos primero nuestro segundo programa:

```

#include <stdio.h>

// Directiva 1: #define para definir una constante
#define MAX 5

// Directiva 2: #ifndef y #define para evitar inclusiones múltiples
#ifndef MENSAJE_Hola
#define MENSAJE_Hola
const char *MENSAJE = "Hola, bienvenido al programa!!\n";
#endif

// Directiva 3:
#pragma message("Compilando el programa con 4 directivas del preprocesador")

// Directiva 4:
#define NUMEROS_O_LETRAS 1
#undef NUMEROS_O_LETRAS
int main(void) {
    printf("%s", MENSAJE);
    #if NUMEROS_O_LETRAS == 1
        printf("Imprimiendo numeros:\n");
        for (int i = 0; i < MAX; i++) {
            printf("%d\n", i);
        }
    #else
        printf("Imprimiendo letras.\n");
        for (int i = 0; i < MAX; i++){
            printf("%c\n", 'a'+i);
        }
    #endif
    return 0;
}

```

a) Explique su utilidad general y su función en particular para su programa.

Veamos primero nuestras cuatro directivas:

- `#define MAX`: Define una constante con el valor de 5.
- `#ifndef MENSAJE_Hola` y `#define MENSAJE_Hola`: Estas directivas evitan inclusiones multiples. La primera vez que se encuentra la definicion se define como "Hola, bienvenido al programa!!", por lo que si ya estaba definido, se ignora la definición siguiente.
- `#pragma message("Compilando el programa con 4 directivas del preprocesador")`: Muestra ese mismo mensaje durante la compilación.
- `#define NUMEROS_O_LETRAS 1` y `#undef NUMEROS_O_LETRAS`: Define la ocntante y luego se desdefine, esto es importante para la función del programa ya que dependiendo si esta o no definida el programa se comportara diferente.

Ahora, su función es dar otro ejemplo de cómo las directivas del preprocesador pueden controlar el flujo de compilación y la ejecución del programa, permitiendo una mayor flexibilidad y control sobre el código.

Veamos que si desdefinimos el `NUMEROS_O_LETRAS` obtendremos entrara en el `else` lo cual nos dara una salida como:

```
martillo@martillo:~/Desktop/Ejercicio$ ./ejecutable
Hola, bienvenido al programa!!
Imprimiendo letras.
a
b
c
d
e
```

Ahora, veamos que si comentamos la línea que desdefine esta definición, obtendremos un resultado diferente, en este caso sera un conteo de numeros usando de igual manera MAX:

```
martillo@martillo:~/Desktop/Ejercicio$ ./ejecutable
Hola, bienvenido al programa!!
Imprimiendo numeros:
0
1
2
3
4
```

b) Redacte un informe detallado con sus resultados y conclusiones.

Este reporte junto con los scripts de cada ejercicio seran adjuntados tanto en classroom como en github para mejor visualización.

Conclusiones

A lo largo de esta practica, vimos de cerca el proceso de compilación en C, este es un flujo de trabajo bien definido que involucra varias etapas, como lo son el preprocesamiento, compilación, ensamblado y enlazado. Las directivas del preprocesador se usan para controlar este proceso, permitiendo definir constantes, incluir archivos y compilar condicionalmente.

Ademas de que cada etapa estuvo marcada por un comando, veamos como fue este procedimiento:

- Preprocesamiento: El preprocesador (cpp) expande macros, incluye archivos de cabecera y elimina comentarios, este nos genero un archivo preprocesado (.i).
- Compilación: El compilador (gcc) traduce el archivo preprocesado (.i) a código ensamblador (.s).
- Ensamblado: El ensamblador (as) convierte el código ensamblador (.s) en código objeto (.o).
- Enlazado: El enlazador (ld) combina los archivos objeto (.o) y las bibliotecas para generar un ejecutable.

Referencias:

- gcc -Wall option flag (compiler warnings). (s.f.). <https://www.rapidtables.com/code/linux/gcc/gcc-wall.html>
- cpp(1) - Linux manual page. (s.f.). <https://www.man7.org/linux/man-pages/man1/cpp.1.html>
- Using the GNU Compiler Collection (GCC). (s.f.). <https://gcc.gnu.org/onlinedocs/gcc-3.2.2/gcc/Option-Summary.html#Option%20Summary>