

Software en formato fuente

Eugenia Damonte, Ariel Fideleff y Martín Goñi

Índice

1. Configuración previa	1
2. Uso básico de gcc	2
2.1. Compilación directa	2
2.2. Compilación compleja	2
2.2.1. Preprocesamiento	3
2.2.2. Compilación	4
2.2.3. Ensamblado	4
2.2.4. Enlazado	5
2.2.4.1. Enlazando en la práctica	5
2.2.4.2. Tipos de enlazado	8
3. Uso básico de make	13
3.1. Bases de make	13
3.2. Compilación simple con make	13
4. Compilando con make	16
5. Comandos usados	23
6. Archivos generados	27

1. Configuración previa

Antes de comenzar a resolver los ejercicios configuramos `vim` para editar archivos en C. Para hacer esto abrimos el archivo `~/.vimrc` (en todo caso de no existir, hay que crearlo) que es el archivo de configuración de `vim`. Estaba vacío por lo que le añadimos dos líneas: `set nocp` y `filetype plugin on`. Lo que hace el primer comando es desactivar el modo de compatibilidad. Éste hace que algunas de las funciones de `vim` sean deshabilitadas o modificadas para que se comporte de manera similar a `vi`, el antecesor de `vim`. La segunda permite utilizar el plugin `filetype`.

Luego para asegurarnos de tener todos los paquetes de `vim` utilizamos el comando `sudo apt-get install vim-gui-common vim-runtime`. El primer paquete tuvo que instalarse y demoró varios minutos por la velocidad de descarga abismal de los repositorios. El segundo, por el otro lado ya estaba instalado en nuestro caso.

Finalmente creamos el archivo de configuración para los archivos con extensión `.c`, llamado `c.vim`. Para poder crearlo primero tuvimos que crear la carpeta `~/.vim/ftplugin`, que es donde se ponen los archivos de configuración. Luego abrimos el mismo con `vim` y escribimos las configuraciones que queríamos usar.

```
setlocal number
syntax on
colorscheme desert
setlocal tabstop=4
setlocal shiftwidth=1
setlocal expandtab
setlocal softtabstop=4
setlocal autoindent
setlocal smartindent
```

Figura 1.1: Las configuraciones para los archivos `.c`

2. Uso básico de gcc

Antes de comenzar con el proyecto en sí, decidimos asegurarnos de que `gcc` funcionaba correctamente y que sabíamos usarlo. Para hacer esto copiamos el programa de ejemplo, `circulo.c`, que se encuentra en el apunte provisto.

```
1 #define PI 3.1416
2
3 main() {
4     float area, radio;
5
6     radio = 10;
7     area = PI * (radio * radio);
8     printf("Circulo.\n");
9     printf("%s%f\n\n", "Área de circulo de radio 10: ", area);
10 }
```

Figura 2.1: El programa de ejemplo `circulo.c`

2.1. Compilación directa

Una vez copiado el programa realizamos una compilación directa para asegurarnos de que funcionase correctamente. Para hacer esto usamos el comando `gcc -o circulo circulo.c`. Lo que hace el argumento `-o` es permitirnos especificar el nombre del archivo de salida, pues si no, el archivo se nombra por defecto `a.out`.

```
martin@DebianPC:~/C$ gcc -o circulo circulo.c
martin@DebianPC:~/C$ ./circulo
Circulo.
Area de circulo de radio 10: 314.160004
martin@DebianPC:~/C$ rm circulo
```

Figura 2.2: Muestra del funcionamiento de `circulo.c`

2.2. Compilación compleja

Habiendo comprobado que `gcc` funcionaba correctamente decidimos intentar compilar el mismo archivo, `circulo.c`, de manera compleja. Es decir, haciendo cada uno de los pasos que realiza el compilador a la hora de transformar un archivo en C en un programa ejecutable, manualmente uno por uno.

2.2.1. Preprocesamiento

El preprocesado o preprocesamiento es la primera etapa de modificación del código fuente. Sirve para que, en la fase de compilación, que es la siguiente, el compilador pueda leer correctamente el código.

El trabajo del preprocesador consiste en llevar a cabo las instrucciones dadas por las *directivas* dirigidas al mismo (que, en el caso de C y C++, son las que comienzan con un numeral, como `#define`, `#include`, `#ifdef` y `#error`, entre otros).

En el archivo `circulo.c` podemos encontrar la directiva `#define PI 3.1416`, que justamente define una constante de nombre `PI` y valor `3.1416`. Esta constante es llamada en la función `main` de manera que, en vez de escribir `3.1416`, escribimos simplemente `PI`.

En la figura 2.3, se observa el código preprocesado que obtuvimos como salida del comando `gcc -E circulo.c` (también se puede usar `cpp circulo.c`, que hace referencia directamente al preprocesador). Si prestamos atención, la directiva antes mencionada no figura en esta salida. Además, en la línea que en el código fuente decía `area = PI * (radio * radio)`, ahora dice `area = 3.1416 * (radio * radio)`.

En resumen, lo que hizo el preprocesador fue tomar esa definición que le indicamos en la directiva y colocar el valor de la constante en las partes del código en donde se hacía referencia a ella.

```
# 1 "circulo.c"
# 1 "<command-line>"
# 1 "circulo.c"

main() {
    float area, radio;

    radio = 10;
    area = 3.1416 * (radio * radio);
    printf("Circulo.\n");
    printf("%s%f\n\n", "Area de circulo de radio 10: ", area);
}
```

Figura 2.3: Resultado del preprocesado de `circulo.c`

2.2.2. Compilación

La compilación es el proceso donde se transforma el código antes pre-procesado (en nuestro caso en C), a assembler propio del procesador de la computadora (en español, *lenguaje ensamblador*). Para hacer esto usamos el comando `gcc -S circulo.c`. Notar que directamente nos referimos al archivo con el código fuente `circulo.c`, pues el argumento `-S` ya de por medio realiza el preprocesado antes explicado. Finalmente verificamos que haya funcionado el comando, mostrando las primeras líneas del archivo `circulo.s`, que es donde `gcc` almacena el compilado.

```
martin@DebianPC:~/C$ gcc -S circulo.c
martin@DebianPC:~/C$ head circulo.s
        .file     "circulo.c"
        .section  .rodata
.LC2:
        .string  "Circulo."
.LC3:
        .string  "Area de circulo de radio 10: "
.LC4:
        .string  "%s%f\n\n"
        .text
        .globl  main
```

Figura 2.4: Primeras 10 líneas del resultado de la compilación

2.2.3. Ensamblado

Una vez realizado el compilado, procedemos a ensamblar el archivo. Es decir, transformar el archivo de assembler a código objeto, un archivo binario en lenguaje máquina. Hicimos esto con el comando `as -o circulo.o circulo.s`. Luego verificamos que haya funcionado revisando qué tipo de archivo era `circulo.o` haciendo uso del comando `file`.

Otra forma por la cual podríamos haber obtenido el archivo en cuestión sería utilizando el comando `gcc -c circulo.c`. Pero, como se puede ver, comenzaría a operar directamente desde el código fuente y realizaría todas las etapas previas explicadas.

```
martin@DebianPC:~/C$ as -o circulo.o circulo.s
martin@DebianPC:~/C$ file circulo.o
circulo.o: ELF 32-bit LSB relocatable, Intel 80386, version 1 (SYSV), not stripped
```

Figura 2.5: Ejecución del ensamblado y detalles del archivo generado

Como el archivo conseguido está en lenguaje máquina, no podemos verlo con facilidad. En todo caso, podemos hacer uso de un comando como lo es `objdump -d circulo.o`, que interpreta y permite ver las instrucciones a la computadora contenidas en el archivo. Hay que aclarar que las posiciones de memoria a las que se refiere probablemente no sean correctas, ya que éstas deben ser relacionadas en el próximo paso, el enlazado, con las librerías externas utilizadas.

2.2.4. Enlazado

2.2.4.1. Enlazando en la práctica

Finalmente enlazamos el archivo. El enlazado es el proceso mediante el cual se vinculan e incorporan al programa las librerías que requiere para poder funcionar. Estas librerías están compuestas de código con funciones que uno utiliza en dicho programa. Por ejemplo, en `circulo.c` usamos funciones como `printf`, que hace referencia a la librería `stdio.h` (típicamente se definiría al comienzo del programa de la forma `#include<stdio.h>` pero, al ser comúnmente utilizado, es incorporado automáticamente por el compilador).

En este paso es donde nos encontramos con problemas. El comando que se da en el apunte no funciona. Al usarlo, reporta varios errores: indica que las librerías usadas como argumentos no existen y que algunas de las opciones del comando no son válidas.

```
martin@DebianPC:~/C$ ld -o circulo /usr/lib/gcc-lib/i386-linux/2.95.2/collect2 -m elf_i386 -dynamic-linker /lib/ld-linux.so.2 -o
circulo /usr/lib/crti.o /usr/lib/crti.o /usr/lib/gcc-lib/i386-linux/2.95.2/crtbegin.o -L/usr/lib/gcc-lib/i386-linux/2.95.2 circ
ulo.o -lgcc -lc -lgcc /usr/lib/gcc-lib/i386-linux/2.95.2/crtend.o /usr/lib/crtn.o
ld: cannot find /usr/lib/gcc-lib/i386-linux/2.95.2/collect2: No such file or directory
ld: cannot find /usr/lib/crti.o: No such file or directory
ld: cannot find /usr/lib/crti.o: No such file or directory
ld: cannot find /usr/lib/gcc-lib/i386-linux/2.95.2/crtbegin.o: No such file or directory
ld: cannot find -lgcc
```

Figura 2.6: El primer intento de usar `ld`, siguiendo el apunte

Dado todos los errores que habían, decidimos borrar todas las opciones innecesarias y probar nuevamente. Al hacer esto, los errores anteriores desaparecieron a cambio de uno nuevo. Éste decía “cannot find entry symbol `_start`”, que se traduce como “no se encuentra el símbolo de entrada `_start`”. Luego de algo de investigar descubrimos que este error se debe a que el verdadero punto de entrada¹ de un programa es `_start` y no `main`, siendo que `_start` simplemente redirige a él. Para solucionar esto, usamos el argumento `--entry main` que establece a la función `main` como el punto

de entrada del programa.

Una vez hechos estos cambios, la función no dio más errores y el programa parecía estar listo para usar. A la hora de ejecutarlo, sin embargo, éste no era reconocido como un programa ejecutable. Para asegurarnos de haber hecho todo correctamente revisamos que el archivo existiese así como también sus permisos, siendo éstos correctos.

```
martin@DebianPC:~/C$ ld -o circulo circulo.o -lc
ld: warning: cannot find entry symbol _start; defaulting to 000000000080481e0
martin@DebianPC:~/C$ ld -o circulo circulo.o -lc --entry main
martin@DebianPC:~/C$ ./circulo
-bash: ./circulo: No such file or directory
martin@DebianPC:~/C$ ls -l circulo
-rwxr-xr-x 1 martin martin 2245 Jul 29 16:20 circulo
```

Figura 2.7: El segundo intento de usar `ld`

Dado que no podíamos ejecutar el programa decidimos intentar volver a añadir algunas de las opciones que no causaban errores. Primero volvimos a añadir las opciones `-m elf_i386` y `--dynamic-linker /lib/ld-linux.so.2`. La primera define el objetivo del compilador² y, la segunda, la ubicación del enlazador dinámico³ a usar.

Luego de hacer estos cambios logramos ejecutar el programa, que parecía funcionar correctamente. Sin embargo, al final de éste tuvimos el error **Segmentation fault**. Para tratar de averiguar de donde venía el error decidimos debuggear el programa utilizando `gdb`.

²El punto de entrada de un programa es donde se ejecutan las primeras instrucciones y se pasa control al programa.

³El objetivo del compilador es lo que determina que tipo de código objeto debe producir la función.

³Un enlazador dinámico o *dynamic linker* es una forma de enlazar los archivos binarios que se necesitan para que el programa funcione. En este caso, el código de las funciones se mantiene en la biblioteca y, a la hora de ejecutar el programa, se carga en memoria.


```

martin@DebianPC:~/C$ ld -o circulo -m elf_i386 --dynamic-linker /lib/ld-linux.so.2 circulo.o -lc --entry main
martin@DebianPC:~/C$ ./circulo
Circulo.
Area de círculo de radio 10: 314.160004
Segmentation fault

```

(a) Error al correr el programa

```

martin@DebianPC:~/C$ gdb circulo
GNU gdb (GDB) 7.4.1-debian
Copyright (C) 2012 Free Software Foundation, Inc.
License GPLv3+: GNU GPL version 3 or later <http://gnu.org/licenses/gpl.html>
This is free software: you are free to change and redistribute it.
There is NO WARRANTY, to the extent permitted by law. Type "show copying"
and "show warranty" for details.
This GDB was configured as "i486-linux-gnu".
For bug reporting instructions, please see:
<http://www.gnu.org/software/gdb/bugs/>...
Reading symbols from /home/martin/C/circulo...(no debugging symbols found)...done.
(gdb) r
Starting program: /home/martin/C/circulo
Circulo.
Area de círculo de radio 10: 314.160004

Program received signal SIGSEGV, Segmentation fault.
0x00000001 in ?? ()

```

(b) Nuestro intento de debuggear el programa

Figura 2.8: El tercer intento de usar ld

Cuando intentamos debuggear el programa nos encontramos con algo extraño, y es que `gdb` no sabía de qué línea provenía el error. Esto nos llevó a creer que provenía del enlazado del programa, y no del programa en sí. Luego de buscar más todavía, encontramos el problema: nos faltaba incluir las librerías que requería el enlazador dinámico. Para entender por qué pasa esto hay que entender cómo funciona el comando.

Lo primero que hace el comando es especificar la ubicación del enlazador dinámico que requieren las demás librerías para acceder a las funciones dinámicas de C. Luego se incluyen otras tres librerías `/usr/lib/i386-linux-gnu/crt1.o`, `/usr/lib/i386-linux-gnu/crti.o` y `/usr/lib/i386-linux-gnu/crtn.o`. La primera es la librería que tiene referencias a los archivos que requiere el enlazador (`/lib/libc.so.6` y `/usr/lib/libc_nonshared.a`). Las otras dos se encargan de que existan `_init` y `_fini`, que son el código de inicialización y finalización. Algo importante de recordar es que la ubicación de las librerías puede cambiar dependiendo del sistema y la instalación específica. En nuestro caso, las encontramos buscando en `/usr/lib` y revisando todas las carpetas que parecían tener alguna relación.

Es importante notar la posición de las librerías, `crti.o` debe ir después de `crt1.o`. Esto es porque el segundo hace referencia al primero. Además

ambas deben ir antes del archivo que se está enlazando. Finalmente `crtn.o` va al final del comando, después de todos los demás argumentos.

```
martin@DebianPC:~/C$ ld -o circulo -m elf_i386 --dynamic-linker /lib/ld-linux.so.2 /usr/lib/i386-linux-gnu/crti.o /usr/lib/i386-linux-gnu/crtn.o circulo.o -lc /usr/lib/i386-linux-gnu/crtn.o
martin@DebianPC:~/C$ ./circulo
Circulo.
Area de circulo de radio 10: 314.160004
martin@DebianPC:~/C$ _
```

Figura 2.9: El cuarto y último intento de usar `ld`

Luego de hacer todo esto, el programa finalmente funcionó y se ejecutó de manera correcta y sin errores.

2.2.4.2. Tipos de enlazado

Ahora bien, como demostramos, en el enlazado “unimos” todos los códigos objetos que forman nuestro programa (pueden ser varios en aplicaciones más complejas) para crear un único archivo ejecutable. Pero no hay una única forma de enlazar, pues existen dos tipos: el enlazado *estático* y el *dinámico* (este último antes brevemente mencionado como *dynamic linker*).

Cuando se hace un enlazado estático, el programa que se está ocupando de realizar el enlazado (en nuestro caso `ld`) incluye en el ejecutable generado las funciones usadas en el programa a enlazar, junto con cualquier dependencia que tengan. Para hacer esto, revisa cada llamada a una función que hay en dicho programa, y si hay una definición de las mismas en él. Si no encuentra ninguna, las busca en las librerías incluidas y las añade al ejecutable final (suponiendo que dichas funciones existan). Un ejemplo es el programa `circulo.c`: éste usa la función `printf`, de la librería `stdio.h`. Si se realiza un enlazado estático, se incluirá la definición de la función con todas sus dependencias dentro del archivo binario generado. Este tipo de enlace es útil cuando se quiere que un programa no tenga dependencias, y previene también problemas debido a cambios en las librerías usadas. Por otro lado, esto hace que el binario ocupe más espacio, y previene resolver bugs (errores) relacionados con las librerías usadas.

En el enlazado dinámico, en cambio, las librerías no son copiadas en el ejecutable, sino que se hace referencia a ellas dinámicamente. Esto significa que dichas librerías deben estar presentes en el sistema al correr el ejecutable, y en dicho momento a su vez se las copia temporalmente en la memoria RAM. En comparación con un enlazado estático, entonces, el tamaño del

archivo ejecutable es mucho menor en lo que se refiere a código objeto del programa, pero sí ocupa más espacio en la memoria RAM durante la ejecución. Esto último permite que las distintas partes del programa que usen una misma función varias veces, hagan referencia a un mismo pedazo de código ya cargado en memoria en vez de leerlo del programa en sí, es decir, evita la repetición de código causada en el enlazado estático.

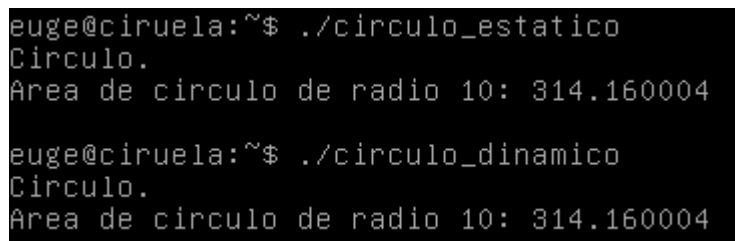
Para probar y experimentar las diferencias entre el enlazado dinámico y el estático, nos pareció una buena idea usar un mismo programa, crear dos ejecutables (uno de cada forma) y compararlos.

El primer programa con el que hicimos esta prueba fue con `circulo.c`, que usamos al principio de este trabajo.

Para enlazarlo dinámicamente, hicimos lo que se muestra previamente en esta sección (nombrando al ejecutable como `circulo_dinamico`), aunque también podríamos haber usado `gcc -o circulo_dinamico circulo.c`, que toma el código fuente y realiza, además del enlazado, todas las etapas anteriores (preprocesado, compilado y ensamblado).

Para enlazarlo estáticamente, usamos un comando parecido al anterior: `gcc --static -o circulo_estatico circulo.c`

Antes de proceder, verificamos que ambos ejecutables funcionaran correctamente (2.10) y que fueran del tipo de archivo que deseábamos. Para esto último, usamos los comandos `file circulo_estatico` y `file circulo_dinamico`, cuyos resultados pueden verse en la imagen 2.11.



```
euge@ciruela:~$ ./circulo_estatico
Circulo.
Area de circulo de radio 10: 314.160004

euge@ciruela:~$ ./circulo_dinamico
Circulo.
Area de circulo de radio 10: 314.160004
```

Figura 2.10: Verificamos que ambos ejecutables anduvieran correctamente

```
euge@ciruela:~$ file circulo_estatico
circulo_estatico: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), statically linked, for GNU/Linux 2.6.26, BuildID[sha1]=0x1dc8641753be3a523fd8d71ada7ebf9f6c51821e, not stripped
euge@ciruela:~$ file circulo_dinamico
circulo_dinamico: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linked (uses shared libs), for GNU/Linux 2.6.26, not stripped
euge@ciruela:~$ _
```

Figura 2.11: Verificamos que los ejecutables estuvieran bien enlazados

Lo siguiente que hicimos fue observar la diferencia de tamaño. En la figura 2.12, podemos ver la salida del listado con detalle de estos dos ejecutables. La quinta columna indica el tamaño que ocupa cada archivo en bytes. `circulo_dinamico`, enlazado dinámicamente, pesa 3649 bytes. Mientras que `circulo_estatico`, enlazado estáticamente, 594184 bytes. Está más que confirmado que el programa con enlazado estático pesa más que el dinámico.

```
euge@ciruela:~$ ls -l circulo_estatico circulo_dinamico
-rwxr-xr-x 1 euge euge 3649 ago 5 21:40 circulo_dinamico
-rwxr-xr-x 1 euge euge 594184 ago 6 21:35 circulo_estatico
euge@ciruela:~$ _
```

Figura 2.12: Observamos la diferencia de tamaño entre los ejecutables

Por último, comparamos cuánta memoria RAM ocupaba la ejecución de cada programa. Sabíamos que el comando `ps` brinda información variada sobre procesos que se están ejecutando en la máquina. Pero, como nuestro programa tardaba tan poco en finalizar, no llegábamos a verlo en la salida de este.

Empezamos a investigar sobre distintas formas que sirven para ver datos de un proceso finalizado o de un archivo que tarda muy poco en ejecutarse: `time -v`, `tstime`, `valgrind...` pero no logramos lo que queríamos hasta que recordamos que los procesos detenidos pueden verse en la salida de `ps`.

Tratamos de detener la ejecución de alguno de los ejecutables de `circulo.c`, pero éste tardaba aproximadamente tres segundos en finalizar, y no lográbamos pararlo. Si conseguíamos un programa que hiciera más cosas (no tantas, pero que tuviera un mayor tiempo de ejecución), íbamos a poder detener el proceso y cumplir con nuestro objetivo.

Por lo tanto, hicimos un programa con un tiempo de ejecución más grande (de aproximadamente doce segundos): `cosas_innecesarias.c` (se puede ver el código en la sección de Archivos generados). El procedimiento es pare-

cido que con `circulo.c`: creamos dos ejecutables (`cos_estaticas` -enlazado estáticamente- y `cos_dinamicas` -enlazado dinámicamente-) y comprobamos que ambos anduvieran.

```
euge@ciruela:~$ gcc --static -o cos_estaticas cosas_innecesarias.c
euge@ciruela:~$ file cos_estaticas
cos_estaticas: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), statically linked, for GNU/Linux 2.6.26, BuildID[sha1]=0x2422b8a15556440fcf1fbf2bffb53e76e8cea071, not stripped
euge@ciruela:~$ gcc -o cos_dinamicas cosas_innecesarias.c
euge@ciruela:~$ file cos_dinamicas
cos_dinamicas: ELF 32-bit LSB executable, Intel 80386, version 1 (SYSV), dynamically linked (uses shared libs), for GNU/Linux 2.6.26, BuildID[sha1]=0x1015fdd7fd915eab7536f06d7c9d9baf4d36c6c6, not stripped
euge@ciruela:~$ _
```

Figura 2.13: Creamos los nuevos ejecutables y verificamos que estuvieran correctamente enlazados

Ejecutamos uno de dichos archivos, detuvimos el proceso con la combinación de teclas `Ctrl+Z` y repetimos lo mismo con el otro. Por último, hicimos uso del comando `ps`, de la forma `ps auxT`. La salida de esa instrucción puede observarse en la imagen 2.15.

```
euge@ciruela:~$ ./cos_dinamicas
Buenas tardes
El numero ganador de la loteria es 16, porque 'Hola, soy Pedrin' tiene esa cantidad de caracteres
^Z
[1]+  Detenido                  ./cos_dinamicas
euge@ciruela:~$ ./cos_estaticas
Buenas tardes
El numero ganador de la loteria es 16, porque 'Hola, soy Pedrin' tiene esa cantidad de caracteres
^Z
[2]+  Detenido                  ./cos_estaticas
euge@ciruela:~$ _
```

Figura 2.14: Detuvimos la ejecución de los dos ejecutables del programa

USER	PID	%CPU	%MEM	VSZ	RSS	TTY	STAT	START	TIME	COMMAND
euge	3299	0.0	0.0	1724	404	tty1	T	22:26	0:00	./cos_dinamicas
euge	3300	0.0	0.0	812	176	tty1	T	22:26	0:00	./cos_estaticas

Figura 2.15: Salida del comando `ps auxT`

Como podemos observar, la salida de este comando tiene varios campos: usuario, identificador del proceso, porcentaje del CPU usado, porcentaje de la memoria utilizada, VSZ, RSS, terminal en donde se está llevando a cabo el proceso, estado (la T indica que los procesos están detenidos), hora de inicio, tiempo que lleva corriendo, comando que lo inició.

Los campos que nos interesan analizar son **VSZ** y **RSS**, relacionados con la memoria RAM. **VSZ** hace referencia a la *Virtual Set Size*, que es la memoria RAM que hay disponible para llevar a cabo un proceso. Para calcularla, la máquina tiene en cuenta toda la memoria a la que el proceso podría llegar a acceder: la que ocupan las bibliotecas dinámicas y la asignada (memoria que ocupan las variables utilizadas en un programa).

$$\text{VSZ} = \text{total bibliotecas dinámicas} + \text{total memoria asignada}$$

Por otro lado, la **RSS** es la memoria RAM que el proceso tiene en uso. Cuando se calcula se tienen en cuenta la memoria que ocupan las partes de las librerías dinámicas que ya fueron cargadas en la RAM y las partes de la memoria asignada que está en uso, es decir, sólo considera los espacios de memoria que contienen información y estén siendo utilizados por el programa.

$$\text{RSS} = \text{bibliotecas dinámicas cargadas} + \text{memoria asignada usada}$$

En la práctica: la **VSZ** de `cos_dinamicas` es de 1724 KB, mientras que la de `cos_estaticas` es de sólo 812 KB. La diferencia está en las bibliotecas dinámicas: al ser el mismo programa, ambos archivos tienen la misma cantidad de memoria asignada, por lo que en lo único que pueden diferir es en la memoria que ocupan es en dichas bibliotecas (que en el caso de `cos_estaticas` es igual a cero porque está enlazado estáticamente).

Gracias a estos datos, podemos confirmar que, al ser ejecutado, un programa enlazado dinámicamente ocupa más memoria RAM que uno enlazado estáticamente.

Habiendo terminado, decidimos que ya teníamos conocimiento suficiente como para intentar compilar un programa usando `make`.

3. Uso básico de make

3.1. Bases de make

El programa **make** es una herramienta que permite manejar y mantener programas que constan de muchos archivos y tienen múltiples dependencias. Éste evita tener que volver a recompilar el programa manualmente cada vez que se hacen cambios. Automáticamente detecta qué archivos necesitan ser recompilados y da las instrucciones para hacerlo, todo con un solo comando.

La base del sistema esta un archivo llamado **makefile**, en el que se almacenan las instrucciones que **make** utiliza a la hora de compilar un programa. Éste contiene las reglas de dependencia, macros y las reglas implícitas. Las reglas de dependencia son reglas que definen qué archivos son necesarios para crear un objetivo⁴. Los macros son variables que almacenan un valor determinado y son muy útiles para evitar repetir argumentos una gran cantidad de veces. Finalmente, las reglas implícitas son una manera de especificar reglas o condiciones usadas a la hora de compilar el programa.

3.2. Compilación simple con make

Para asegurarnos de que sabíamos usar **make**, decidimos compilar el programa de ejemplo que aparece en el apunte, la calculadora con notación polaca. Primero tuvimos que llevar los archivos a la máquina virtual, por lo que los descargamos de la página web mencionada en el apunte utilizando **wget**.

```
martin@DebianPC:~/C/calculadora_polaca$ wget -q -O calc.h ie.fing.edu.uy/~vagonbar/gcc-make/polaca/calc.h
martin@DebianPC:~/C/calculadora_polaca$ wget -q -O getch.c ie.fing.edu.uy/~vagonbar/gcc-make/polaca/getch.c
martin@DebianPC:~/C/calculadora_polaca$ wget -q -O getop.c ie.fing.edu.uy/~vagonbar/gcc-make/polaca/getop.c
martin@DebianPC:~/C/calculadora_polaca$ wget -q -O main.c ie.fing.edu.uy/~vagonbar/gcc-make/polaca/main.c
martin@DebianPC:~/C/calculadora_polaca$ wget -q -O stack.c ie.fing.edu.uy/~vagonbar/gcc-make/polaca/stack.c
martin@DebianPC:~/C/calculadora_polaca$ ls
calc.h  getch.c  getop.c  main.c  stack.c
```

Figura 3.1: Descargamos los archivos a la VM

Una vez descargados todos los archivos, compilamos el programa sin utilizar **make** para asegurarnos de que éste funcionase correctamente. Para hacer

⁴Un objetivo es el nombre de un archivo(ejecutable o objeto) o una acción de ese **makefile**.

esto usamos `gcc`, dándole como argumento todos los archivos que necesitaban ser compilados.

```
martin@DebianPC:~/C/calculadora_polaca$ gcc -o main main.c getch.c getop.c stack.c
stack.c: In function 'push':
stack.c:17:5: warning: incompatible implicit declaration of built-in function 'exit' [enabled by default]
martin@DebianPC:~/C/calculadora_polaca$ ./main

CALCULADORA POLACA.
Ingrese expresiones en notación polaca inversa.
Para finalizar, digite Ctrl-D.
8 5 *
Resultado:      40
```

Figura 3.2: Compilamos el programa sin `make`

Habiendo confirmado que el programa funcionaba, creamos el archivo `makefile`, necesario para que `make` funcione. En éste colocamos todas las reglas necesarias para compilar el programa. Éstas especificaban qué se debía hacer con cada archivo. Para hacer esto usamos reglas de dependencia, las cuales siguen la siguiente estructura:

```
destino : dependencias
comando
```

Donde `destino` es el nombre del archivo a donde irá el resultado de las acciones de `comando`. Es importante notar que `destino` también puede ser una acción que puede ser llamada desde el archivo o al usar `make`. Luego tenemos `dependencias`, que son los archivos que se usan para crear `destino` y deben estar presentes en la ubicación especificada. Por último, en `comando` se indican una o varias instrucciones que son enviadas al shell para ser interpretadas.


```
OBJECTS = stack.o getop.o getch.o
CC = gcc

polaca: main.o $(OBJECTS)
    $(CC) -o polaca main.o $(OBJECTS)

main.o: main.c calc.h
    $(CC) -c main.c
stack.o: stack.c calc.h
    $(CC) -c stack.c
getop.o: getop.c calc.h
    $(CC) -c getop.c
getch.o: getch.c
    $(CC) -c getch.c

clean:
    -rm polaca main.o $(OBJECTS)
```

Figura 3.3: El `makefile` creado para compilar el programa

Sabiendo cómo crear reglas de dependencia y con ayuda del apunte, creamos el `makefile` necesario para compilar el programa. Tiene además una acción para eliminar el archivo ejecutado y todos los archivos `.o` generados por la compilación.

Usando el comando `make polaca`, compilamos el programa exitosamente y sin errores. Para verificar que todo funcionase, realizamos la misma operación que habíamos utilizado al compilar el programa la primera vez. Al ver que los resultados eran iguales y el programa era ejecutado correctamente, dimos por terminada la práctica con `make`. Antes de pasar a compilar `w3m`, borramos todos los archivos innecesarios con `make clean`.

```

martin@DebianPC:~/C/calculadora_polaca$ make polaca
gcc -c main.c
gcc -c stack.c
stack.c:17:5: warning: incompatible implicit declaration of built-in function 'exit' [enabled by default]
gcc -c getop.c
gcc -c getch.c
gcc -o polaca main.o stack.o getop.o getch.o
martin@DebianPC:~/C/calculadora_polaca$ ./polaca

CALCULADORA POLACA.
Ingrese expresiones en notaci#n polaca inversa.
Para finalizar, digite Ctrl-D.
8 5 *
Resultado:      40

```

(a) Compilamos y ejecutamos el programa

```

martin@DebianPC:~/C/calculadora_polaca$ ls
calc.h  getch.c  getch.o  getop.c  getop.o  main.c  main.o  makefile  polaca  stack.c  stack.o
martin@DebianPC:~/C/calculadora_polaca$ make clean
rm polaca main.o stack.o getop.o getch.o
martin@DebianPC:~/C/calculadora_polaca$ ls
calc.h  getch.c  getop.c  main.c  makefile  stack.c

```

(b) Borrarnos todos los archivos innecesarios

Figura 3.4: El comando **make** en funcionamiento

4. Compilando con make

Para poder llevar más a la práctica el conocimiento del funcionamiento de **make** y la compilación de programas mediante makefiles, decidimos compilar **w3m**, un navegador web que funciona dentro de la terminal.

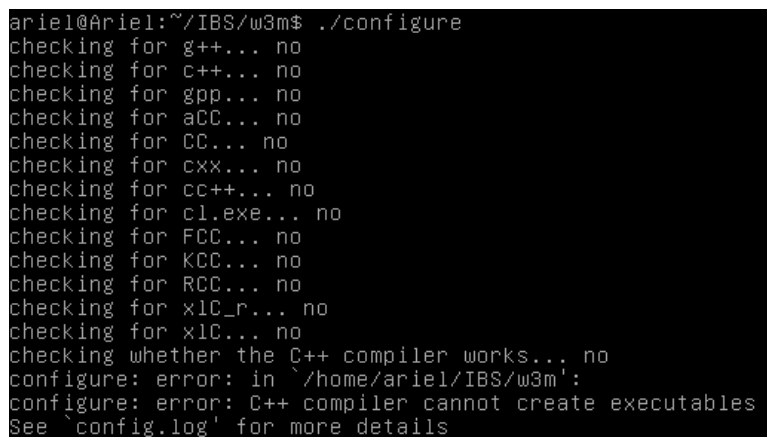
Descargamos el código fuente⁵, los archivos necesarios para el proceso, y notamos que se encontraban en un archivo del tipo **.tar.gz**⁶. Para extraer sus contenidos, usamos el comando **tar** de linux, aunque para ello primero debimos instalarlo usando **sudo apt-get install tar**. Hecho esto, el comando que utilizamos fue **tar -xf w3m-0.5.3.tar.gz**, donde **-x** implica extraer los archivos, y **-f** dice que se ejecuten las acciones sobre un archivo cuya ruta se encuentra a continuación de la opción. Por el nombre del archivo, es claro que la versión de **w3m** descargada es la 0.5.3. Al descomprimir, nos quedó en el lugar de extracción, una carpeta llamada **w3m-0.5.3** con todos los archivos.

Al abrir la carpeta, vimos la presencia de distintos archivos, entre ellos, pu-

⁵<https://sourceforge.net/projects/w3m/files/>

⁶Un archivo **.tar** es utilizado para almacenar múltiples archivos y directorios en un solo archivo. Por otro lado, el sufijo **.gz**, muestra que los contenidos de dicho archivo se encuentran comprimidos y la forma en que fueron comprimidos.

dimos notar algunos de extensión `.c` o `.h`, que, como sabemos, corresponden a código hecho en el lenguaje C. Para saber cómo proceder, nos dirigimos como es habitual al archivo `README` presente, que decía que debíamos dirigirnos a la subcarpeta `doc` para ver las instrucciones en inglés. Allí se encontraba otro `README` con una breve descripción del programa y las instrucciones dichas. Indicaba que primero ejecutáramos el archivo `configure`, presente en la carpeta raíz, de la forma `./configure`. Aquí empezaron los problemas.



```
ariel@Ariel:~/IBS/w3m$ ./configure
checking for g++... no
checking for c++... no
checking for gpp... no
checking for aCC... no
checking for CC... no
checking for cxx... no
checking for cc++... no
checking for cl.exe... no
checking for FCC... no
checking for KCC... no
checking for RCC... no
checking for xlc_r... no
checking for xlc... no
checking whether the C++ compiler works... no
configure: error: in `/home/ariel/IBS/w3m':
configure: error: C++ compiler cannot create executables
See `config.log' for more details
```

Figura 4.1: Primer intento de ejecución de `configure`

Al ejecutar `configure`, se reportó el error “C++ compiler cannot create executables”, es decir, el compilador de C++ no puede crear ejecutables. También nos informó que para más detalles podíamos revisar el archivo generado `config.log`. Allí pudimos ver todo el historial de las acciones realizadas por `configure`, y notamos que verificaba por la presencia de ciertos programas, que parecían ser todos compiladores de C++, implicando que no existía ninguno en la máquina.

```

## ----- ##
## Core tests. ##
## ----- ##

configure:2649: checking for g++
configure:2679: result: no
configure:2649: checking for c++
configure:2679: result: no
configure:2649: checking for gpp
configure:2679: result: no
configure:2649: checking for aCC
configure:2679: result: no
configure:2649: checking for CC
configure:2679: result: no
configure:2649: checking for cxx
configure:2679: result: no
configure:2649: checking for cc++
configure:2679: result: no
configure:2649: checking for cl.exe
configure:2679: result: no
configure:2649: checking for FCC
configure:2679: result: no
:

```

Figura 4.2: Contenidos de `configure.log` luego del primer intento de ejecución de `configure`

Instalamos entonces el primero de entre los que probaba, `g++`, de la forma `sudo apt-get install g++`. Esta vez, al correr `configure` pareció lograr mayor progreso que en el anterior intento pero, de todas formas, la ejecución presentaba un error: `configure: error: gc.h not found`.

```

checking GC library exists... yes
checking gc.h usability... no
checking gc.h presence... no
checking for gc.h... no
checking GC header location... /usr /usr/local /home/ariel
checking /usr/include... checking gc.h usability... no
checking gc.h presence... no
checking for gc.h... no
checking /usr/include/gc... checking gc.h usability... no
checking gc.h presence... no
checking for gc.h... no
checking /usr/local/include... checking gc.h usability... no
checking gc.h presence... no
checking for gc.h... no
checking /usr/local/include/gc... checking gc.h usability... no
checking gc.h presence... no
checking for gc.h... no
checking /home/ariel/include... checking gc.h usability... no
checking gc.h presence... no
checking for gc.h... no
checking /home/ariel/include/gc... checking gc.h usability... no
checking gc.h presence... no
checking for gc.h... no
configure: error: gc.h not found

```

Figura 4.3: Segundo intento de ejecución de `configure`

Estuvimos investigando este problema, viendo cómo se podía obtener este archivo faltante. Notamos que verificaba por la existencia del archivo mencionado en múltiples rutas antes de mostrar el error. Luego de explorar un poco, entendimos que lo que buscaba el programa era algo llamado “Garbage Collector”, que es lo que usan múltiples lenguajes de programación como mecanismo para la gestión adecuada de la memoria, y así tanto reservar espacios de dicha memoria, liberarlos, tener cuenta del espacio libre y ocupado, e incluso la reorganización del mismo a fin de liberar el mayor espacio posible que pueda ser utilizado, compactando los espacios de memoria libres entre los ocupados, dejando la mayor cantidad contigua libre posible.

Después de saltar mucho entre páginas buscando, llegamos a una con información detallada sobre las dependencias que uno puede instalar en Linux, en específico la de `libgc-dev`⁷ ya que vimos que contiene entre sus archivos, el archivo `gc.h` faltante. Lo instalamos de la forma `sudo apt-get install libgc-dev`.

Probamos nuevamente correr `configure`, y por lo que vimos, pareció finalizar sin inconvenientes.

```
config.status: creating scripts/Makefile
config.status: creating scripts/dir1list.cgi
config.status: creating scripts/w3mhelp.cgi
config.status: creating scripts/w3mmail.cgi
config.status: creating scripts/xface2xpm
config.status: creating scripts/multipart/Makefile
config.status: creating scripts/multipart/multipart.cgi
config.status: creating scripts/w3mman/Makefile
config.status: creating scripts/w3mman/w3mman
config.status: creating scripts/w3mman/w3mman.1
config.status: creating scripts/w3mman/w3mman2html.cgi
config.status: creating libwc/Makefile
config.status: creating w3mimg/Makefile
config.status: creating w3mimg/fb/Makefile
config.status: creating w3mimg/x11/Makefile
config.status: creating w3mimg/win/Makefile
config.status: creating w3mhelp-w3m_en.html
config.status: creating w3mhelp-w3m_ja.html
config.status: creating w3mhelp-lynx_en.html
config.status: creating w3mhelp-lynx_ja.html
config.status: creating config.h
config.status: executing po-directories commands
config.status: creating po/POTFILES
config.status: creating po/Makefile
```

Figura 4.4: Tercer intento (y correcto) de ejecución de `configure`

Dado esto, pudimos proceder al siguiente paso: correr el `makefile` para compilar, usando el comando `make` en el directorio raíz. Y como era de

⁷https://ubuntu.pkgs.org/16.04/ubuntu-main-amd64/libgc-dev_7.4.2-7.3_amd64.deb.html

esperarse, hubo problemas también en este paso.

```
/home/ariel/IBS/w3m/terms.c:461: undefined reference to `tputs'
terms.o: In function `setlinescols':
/home/ariel/IBS/w3m/terms.c:822: undefined reference to `tgetnum'
/home/ariel/IBS/w3m/terms.c:824: undefined reference to `tgetnum'
terms.o: In function `refresh':
/home/ariel/IBS/w3m/terms.c:1326: undefined reference to `tgoto'
/home/ariel/IBS/w3m/terms.c:1382: undefined reference to `tgoto'
/home/ariel/IBS/w3m/terms.c:1458: undefined reference to `tgoto'
/home/ariel/IBS/w3m/terms.c:1332: undefined reference to `tgoto'
terms.o: In function `getTcstr':
/home/ariel/IBS/w3m/terms.c:726: undefined reference to `tgetent'
/home/ariel/IBS/w3m/terms.c:733: undefined reference to `tgetstr'
/home/ariel/IBS/w3m/terms.c:734: undefined reference to `tgetstr'
/home/ariel/IBS/w3m/terms.c:735: undefined reference to `tgetstr'
/home/ariel/IBS/w3m/terms.c:738: undefined reference to `tgetflag'
/home/ariel/IBS/w3m/terms.c:747: undefined reference to `tgetstr'
/home/ariel/IBS/w3m/terms.c:748: undefined reference to `tgetstr'
/home/ariel/IBS/w3m/terms.c:749: undefined reference to `tgetstr'
/home/ariel/IBS/w3m/terms.c:750: undefined reference to `tgetstr'
/home/ariel/IBS/w3m/terms.c:751: undefined reference to `tgetstr'
terms.o:/home/ariel/IBS/w3m/terms.c:752: more undefined references to `tgetstr'
follow
collect2: error: ld returned 1 exit status
make: *** [w3m] Error 1
ariel@Ariel:~/IBS/w3m$
```

Figura 4.5: Primer intento de ejecución del makefile

El compilador no pudo encontrar las referencias de una serie de funciones utilizadas como `tputstring` y otros con nombres similares. Luego de buscar un poco, averiguamos que éstas podrían estar presentes en el paquete `libtinfo-dev`, el cual instalamos de la forma `sudo apt-get install libtinfo-dev`. Probamos nuevamente correr `make`.

De todas formas, dada la rapidez con la que corrió en comparación a la anterior ejecución, sumado a que se presentaron los mismos errores que dicha vez, probamos en cambio primero probar con correr `configure` otra vez, y luego `make`.

Comprobamos que el error antes experimentado desapareció, a cambio de otro nuevo: `cannot stat 't-ja.gmo': No such file or directory`. Nuevamente investigamos este error y encontramos en un foro que un error idéntico se le presentó a un usuario cuando quiso instalar otro paquete⁸, y que se podía resolver instalando el paquete `gettext`, de la forma `sudo apt-get install gettext`. Una vez instalado, al igual que antes, volvimos a correr `configure` y `make`.

⁸<https://www.linuxquestions.org/questions/linux-general-1/alsa-utils-failing-make-stage-cannot-stat-%60t-ja-gmo%27-i-think-i-need-xgettext-386546/>

```

make[2]: Entering directory `/home/ariel/IBS/w3m/scripts/w3mman'
make[2]: Nothing to be done for `all'.
make[2]: Leaving directory `/home/ariel/IBS/w3m/scripts/w3mman'
for subdir in multipart w3mman; \
do \
    (cd $subdir && make); \
done
make[2]: Entering directory `/home/ariel/IBS/w3m/scripts/multipart'
make[2]: Nothing to be done for `all'.
make[2]: Leaving directory `/home/ariel/IBS/w3m/scripts/multipart'
make[2]: Entering directory `/home/ariel/IBS/w3m/scripts/w3mman'
make[2]: Nothing to be done for `all'.
make[2]: Leaving directory `/home/ariel/IBS/w3m/scripts/w3mman'
make[1]: Leaving directory `/home/ariel/IBS/w3m/scripts'
(cd po && make)
make[1]: Entering directory `/home/ariel/IBS/w3m/po'
test ! -f ./w3m.pot || \
    test -z "ja.gmo" || make ja.gmo
make[2]: Entering directory `/home/ariel/IBS/w3m/po'
rm -f ja.gmo && /usr/bin/msgfmt -c --statistics --verbose -o ja.gmo ja.po
ja.po: 214 translated messages.
make[2]: Leaving directory `/home/ariel/IBS/w3m/po'
touch stamp-po
make[1]: Leaving directory `/home/ariel/IBS/w3m/po'

```

Figura 4.6: Segundo intento (y correcto) de ejecución del makefile

Esta vez no hubo ningún error visible. Procedimos al paso final, a lo que corrimos `make install`.

```

(cd w3mimg && make CC="gcc" OPTS="")
make[1]: Entering directory `/home/ariel/IBS/w3m/w3mimg'
make[1]: Nothing to be done for `all'.
make[1]: Leaving directory `/home/ariel/IBS/w3m/w3mimg'
mkdir -p /usr/local/bin
mkdir -p /usr/local/libexec/w3m
mkdir: cannot create directory `/usr/local/libexec': Permission denied
make: [install-core] Error 1 (ignored)
mkdir -p /usr/local/libexec/w3m/cgi-bin
mkdir: cannot create directory `/usr/local/libexec': Permission denied
make: [install-core] Error 1 (ignored)
mkdir -p /usr/local/share/w3m
mkdir: cannot create directory `/usr/local/share/w3m': Permission denied
make: [install-core] Error 1 (ignored)
mkdir -p /usr/local/share/man/man1
mkdir: cannot create directory `/usr/local/share/man/man1': Permission denied
make: [install-core] Error 1 (ignored)
mkdir -p /usr/local/share/man/ja/man1
mkdir: cannot create directory `/usr/local/share/man/ja': Permission denied
make: [install-core] Error 1 (ignored)
/usr/bin/install -c w3m /usr/local/bin/w3m
/usr/bin/install: cannot create regular file `/usr/local/bin/w3m': Permission denied
make: *** [install-core] Error 1
ariel@Ariel:~/IBS/w3m$

```

Figura 4.7: Primer intento fallido de ejecución de `make install`

Experimentamos una serie de `Permission denied`, y dedujimos que lo más probable era que, para correr este comando, requeriáramos de permisos `root`. Por lo tanto, probamos agregar `sudo`, al comienzo del comando.

```

done
make[2]: Leaving directory `/home/ariel/IBS/w3m/scripts/w3mman'
make[1]: Leaving directory `/home/ariel/IBS/w3m/scripts'
NLSTARGET="po"; for subdir in $NLSTARGET; \
do \
(cd $subdir && make install); \
done
make[1]: Entering directory `/home/ariel/IBS/w3m/po'
installing ja.gmo as /usr/local/share/locale/ja/LC_MESSAGES/w3m.mo
if test "w3m" = "gettext-tools"; then \
/bin/mkdir -p /usr/local/share/gettext/po; \
for file in Makefile.in.in remove-potcdate.sin quot.sed boldquot.sed a
n@quot.header en@boldquot.header insert-header.sin Rules-quot Makevars.templat
e; do \
/usr/bin/install -c -m 644 ./file \
/usr/local/share/gettext/po/$file; \
done; \
for file in Makevars; do \
rm -f /usr/local/share/gettext/po/$file; \
done; \
else \
: ; \
fi
make[1]: Leaving directory `/home/ariel/IBS/w3m/po'
ariel@Ariel:~/IBS/w3m$

```

Figura 4.8: Segundo intento (y correcto) de ejecución de `make install`

Ningún error visible. Éste habría sido el último paso de la instalación, y deberíamos ser capaces de correr `w3m` correctamente. Para probar su funcionamiento, tratamos de entrar a `www.google.com` con el comando `w3m www.google.com`.

```

Búsqueda Imágenes Maps Play YouTube Noticias Gmail Drive Más >>
Historial web | Preferencias | Iniciar sesión

Google

[ (Buscar con Google) (Me siento con suerte) ] Búsqueda avanzada

Ofrecido por Google en: English
Programas de publicidad Soluciones Empresariales Todo acerca de Google
Google.com.ar

(C) 2020 - Privacidad - Condiciones

« + Viewing <Google>

```

Figura 4.9: Entrando a `www.google.com` con `w3m`

¡Y efectivamente pudimos visualizar el sitio web sin problemas! La compilación de `w3m` fue exitosa.

5. Comandos usados

A continuación se encuentran todos los comandos utilizados en este trabajo, correspondientes a las imágenes presentadas.

```
setlocal number  
syntax on  
colorscheme desert  
setlocal tabstop=4  
setlocal shiftwidth=1  
setlocal expandtab  
setlocal softtabstop=4  
setlocal autoindent  
setlocal smartindent
```

[1.1]

```
gcc -o circulo circulo.c  
./circulo  
rm circulo
```

[2.2]

```
gcc -S circulo.c  
head circulo.s
```

[2.4]

```
as -o circulo.o circulo.s  
file circulo.o
```

[2.5]

```
ld      -o circulo /usr/lib/gcc-lib/i386-linux/2.95.2/
collect2 -m elf_i386 --dynamic-linker
/lib/ld-linux.so.2 -o circulo /usr/lib/crt1.o
/usr/lib/crti.o /usr/lib/gcc-lib/i386-linux/2.95.2/
crtbegin.o -L /usr/lib/gcc-lib/i386-linux/2.95.2
circulo.o -lgcc -lc -lgcc /usr/lib/gcc-lib/
i386-linux/2.95.2/crtend.o /usr/lib/crtn.o
```

[2.6]

```
ld -o circulo circulo.o -lc
ld -o circulo circulo.o -lc --entry main
./circulo
ld -l circulo
```

[2.7]

```
ld      -o circulo -m elf_i386 --dynamic-linker
/lib/ld-linux.so.2 circulo.o -lc --entry main
./circulo
gdb circulo
(gdb) r
```

[2.8]

```
ld -o circulo -m elf_i386 --dynamic-linker
/lib/ld-linux.so.2 /usr/lib/i386-linux-gnu/ crt1.o
/usr/lib/i386-linux-gnu/crti.o      circulo.o      -lc
/usr/lib/i386-linux-gnu/crtn.o
./circulo
```

[2.9]

```
gcc --static -o circulo_estatico circulo.c
./circulo_estatico

gcc -o circulo_dinamico circulo.c
./circulo_dinamico
```

[2.10]

```
file circulo_estatico
file circulo_dinamico
```

[2.11]

```
ls -l circulo_estatico circulo_dinamico
```

[2.12]

```
gcc --static -o cos_estaticas cosas_innecesarias.c
file cos_estaticas

gcc -o cos_dinamicas cosas_innecesarias.c
file cos_dinamicas
```

[2.13]

```
./cos_estaticas
[Ctrl + Z]

./cos_dinamicas
[Ctrl + Z]
```

[2.14]

```
ps auxT
```

[2.15]

```
wget -q -O calc.h iie.fing.edu.uy/~vagonbar/gcc-make/  
polaca/calc.h  
wget -q -O getch.c iie.fing.edu.uy/~vagonbar/gcc-make/  
polaca/getch.c  
wget -q -O getop.c iie.fing.edu.uy/~vagonbar/gcc-make/  
polaca/getop.c  
wget -q -O main.c iie.fing.edu.uy/~vagonbar/gcc-make/  
polaca/main.c  
wget -q -O stack.c iie.fing.edu.uy/~vagonbar/gcc-make/  
polaca/stack.c
```

[3.1]

```
gcc -o main main.c getch.c getop.c stack.c
```

[3.2]

```
make polaca  
make clean
```

[3.4]

```
./configure
```

[4.1]

```
sudo apt-get install gettext  
./configure  
make
```

[4.6]

```
make install
```

[4.7]

```
sudo make install
```

[4.8]

```
w3m www.google.com
```

[4.9]

6. Archivos generados

A continuación se encuentran todos los archivos que creamos para hacer este trabajo, correspondientes a las imágenes presentadas.

```
1 #define PI 3.1416
2
3 main() {
4     float area, radio;
5
6     radio = 10;
7     area = PI * (radio * radio);
8     printf("Circulo.\n");
9     printf("%s%f\n\n", "Area de un circulo de radio 10: ", area);
10 }
```

[2.1] Archivo circulo.c

```
1 #include <stdio.h>
2 #include <string.h>
3 #include <stdlib.h>
4 #include <time.h>
5 #ifdef _WIN32
6     #include <Windows.h>
7     #define suspenso(a) Sleep(a);
8 #else
9     #include <unistd.h>
10    #define suspenso(a) usleep(a*1000);
11 #endif
12
13 /*
14  ADVERTENCIA:
15      Este programa hace unicamente cosas innecesarias.
16      No le busque la vuelta.
```

```

17 */
18
19 struct colores{
20     int a, b;
21     char c[100];
22 };
23
24 int main (){
25
26     struct colores color_es;
27
28     color_es.a = 1;
29     strcpy(color_es.c, "Me gustan los trenes");
30
31     if(color_es.a == 1)
32         printf("Buenas tardes\n");
33
34     srand(time(NULL));
35     int pedrin_o_tal_vez_no = rand() % 31, num;
36
37     char pedrin[20]={'H','o','l','a','',' ','s','o','y','','P',
38                     ',','e','d','r','i','n','\0'};
39
40     int largo_pedrin = 0;
41
42     for(int i = 0; pedrin[i] != '\0'; i++)
43         largo_pedrin++;
44
45     int esto_es_en_vano = strcmp("Que feo el comando ld",
46                                  pedrin);
47
48     printf("El numero ganador de la loteria es %d, porque '%s'
49           tiene esa cantidad de caracteres\n", largo_pedrin, pedrin);
50     suspenso(4000);
51
52     printf("Y tu numero es...\n");
53     suspenso(1000);
54
55     printf("es...\n");
56     suspenso(100);
57
58     printf("El numero que te toco es...\n");
59     suspenso(2000);
60
61     printf("...\n");
62     suspenso(500);
63
64     printf("chan chan\n");
65     suspenso(500);

```

```

63
64     printf("Soy Jorge Suspenso\n");
65     suspenso(1000);
66
67     printf("TU NUMERO ES\n");
68     suspenso(3000);
69
70     printf("%d!!!!\n", pedrin_o_tal_vez_no);
71     suspenso(500);
72
73     if(pedrin_o_tal_vez_no == largo_pedrin)
74         printf("Ganaste la loteria!!!!!! :D\n");
75     else
76         printf("Perdiste, no toda la vida es de color rosa...\n
77                ");
78 return 0;
79 }

```

[2.2.4.2] Archivo cosas_innecesarias.c

```

1 OBJETCS = stack.o getop.o getch.o
2 CC = gcc
3
4 polaca: main.o $(OBJETCS)
5     $(CC) -o polaca main.o $(OBJETCS)
6
7
8 main.o : main.c calc.h
9     $(CC) -c main.c
10 stack.o : stack.c calc.h
11     $(CC) -c stack.c
12 getop.o : getop.c calc.h
13     $(CC) -c getop.c
14 getch.o : getch.c calc.h
15     $(CC) -c getch.c
16
17
18 clean :
19     -rm polaca main.o $(OBJETCS)

```

[3.3] Makefile de polaca.