Laboratorio de
Sistemas Operativos

Práctica 6: Make

Autor: Dr. Juan Arturo Nolazco; **Co-Autores**: M.C. Jorge Villaseñor, M.C. Roberto Aceves, Ing. Raul Fuentes, Dr. Jose I. Icaza

Índice

Objetivos

Esta práctica se enfoca en la administración de proyectos de programación que cuenten con múltiples pequeños módulos y bibliotecas, posiblemente desarrollados por programadores en diversas partes, que deben integrarse para formar un sistema completo.

Probablemente conozcas el uso de herramientas como Visual Studio o Netbeans, entre otras, que permiten trabajar con múltiples archivos fuentes y compilar todos ellos en un único gran proyecto o archivo ejecutable. Pues bien, estas herramientas con interfaces gráficas heredan su funcionalidad de herramientas similares nacidas hace más de una década para la interfaz de línea de comandos. **MAKE** es una de estas herramientas y una de las más poderosas también.

En esta práctica aprenderás a utilizar dicha herramienta de forma básica para compilar múltiples rutinas, tal vez desarrolladas por distintos programadores y que dependen unas de otras; además, tu resultado final podrá incluir librerías estáticas de rutinas previamente compiladas.

Herramienta make:

Cuando se está desarrollando una aplicación el objetivo final es crear un solo archivo ejecutable. Sin embargo, es conveniente *dividir* el trabajo en *pequeños módulos*. Estos módulos finalmente se tendrían que integrar para formar el ejecutable, y pueden ser archivos en un lenguaje particular (i.e. Lenguaje C), o pueden ser códigos binarios (e.g. código objeto o bibliotecas de código objeto tales como stdio, que incluye varios códigos objeto para realizar input/output desde C).En sistemas basados en Unix es muy común utilizar Make para organizar proyectos compuestos de múltiples módulos.

Make es una herramienta con la que se puede mantener un grupo de códigos fuentes, códigos objeto y bibliotecas y compilarlos de modo estructurado. Esta herramienta se encarga de ver qué piezas de software necesitan ser re-compiladas cuando alguna de ellas cambia y de integrar todos los módulos.

EL comando **`make` ejecuta los comandos que se encuentran en el archivo Makefile en la carpeta donde es invocado**.

Empezamos con un ejemplo sencillo de un Makefile, antes de indicar el formato completo de este archivo:

```
makefile:
arch.o: arch.c
<tab>cc -c arch.c -o arch.o
```

Esto nos dice:

Primera línea: Para lograr el objetivo arch.o (es decir, para poder obtener arch.o), es necesario que antes exista su versión fuente, arch.c. O sea, arch.o *depende de* la existencia previa de arch.c.

Segunda línea: “Una vez que exista arch.c, se ejecuta el comando cc arch.c (es decir, se compila arch.c. Eso nos produce arch.o”.

Además, el comando make se encarga de que: “si llega a cambiar arch.c, para obtener arch.o recompila arch.c. Si no ha cambiado arch.c, no es necesario recompilarlo”.

En este caso el comando se invoca simplemente con “make” sin ninguna otra opción. El comando entonces investiga el Makefile (así se debe llamar) y hace lo que corresponda.

El formato más completo del Makefile es el siguiente:

```
objetivo1: dependencias1
<tab>comandos1

objetivo2: dependencias2
<tab>comandos2
(...)
```

***NOTA**: TAB es un tabulador, no 5 espacios en blanco. Si se ponen espacios en vez de TAB, se tendrán problemas de sintaxis*.

Los campos denominados “objetivos” son aquellos campos que Makefile puede ejecutar por separado si se introducen como argumento del comando make. Ejemplo: “***make objetivo2***” ejecutará exclusivamente el objetivo “objetivo2”. En el ejemplo anterior, podríamos entonces decir “make arch.o”, o simplemente “make”, pues toma por default el primer objetivo.. Ahora, las

dependencias son otros archivos u objetivos que deben ya existir sin cambio o que se deben de ejecutar antes de poder realizar el objetivo en cuestión. Los comandos son por lo regular comandos para compilar uno o más códigos fuentes, incorporar códigos objetos, o crear o incorporar bibliotecas dinámicas o estáticas, entre otros.

Si en el ejemplo, después modificamos arch.c y simplemente ejecutamos el comando "**make arch.o**", este comando make se da cuenta de que arch.c fué modificado, y recompila. Esto es debido a que el comando **make** es capaz de detectar si un objetivo ya fue cumplido o si sus dependencias fueron modificadas y necesitan re-evaluarse; de forma interna, lleva un registro de estos cambios. De tal forma que si se tiene un proyecto grande cuya compilación completa tardaría mucho tiempo, "Make" tratará de ahorrar recursos al no re-evaluar objetivos ya cumplidos (Si sus dependencias no han sufrido cambios).

Existe un objetivo en particular denominado "all" y cuando el comando **make** se invoca sin argumentos asume que debe ejecutar dicho objetivo**.** Si "all" no existe (no lo definió el usuario) entonces evalúa el primer objetivo definido en el archivo Makefile. Recuerda que en el momento en que los objetivos son evaluados, se procede a verificar sus dependencias; estas pueden ser nombres de archivos u otros objetivos. Nuevamente regresando al ejemplo, podríamos simplemente haber dicho "make" en vez de "make arch.o" pues solo teníamos ese objetivo.

El archivo Makefile se utiliza para estructurar las dependencias entre módulos de tal modo que sólo se vuelvan a compilar aquellos módulos que hayan sido modificados.

***Curiosidad**: Makefile es ampliamente utilizado; de hecho si has tenido oportunidad de bajar a Linux el código fuente de alguna aplicación para instalarla de forma manual, entonces uno de los pasos que seguramente debiste seguir habrá sido el uso de make, seguramente con "**make [all], o make install**" que se encargan de compilar todo los archivos fuentes y las modificaciones requeridas de los registros de tu sistema operativo.*

Estructura típica de un proyecto

Durante esta práctica crearemos diferentes proyectos los cuales tendrán una estructura de directorios similar a la siguiente:

```
Proyecto
| - bin
| - doc
| - objetos
| - sources
| - misIncludes
```

El objetivo de utilizar esta estructura es acostumbrarte a una separación de piezas de los proyectos que estés generando. Cada carpeta tiene una utilidad especial (y lo puedes comprobar en cualquier proyecto que instales de forma manual) y es la siguiente:

- **bin** - Todos los archivos ejecutables (**bin**arios) que tenga el proyecto son depositados en este folder.
- **doc** - Toda la **doc**umentación que se libere del proyecto suele ser depositada en este directorio.
- **sources** - El código fuente del proyecto (sea un archivo, 10, 100, 500, etc) iría en este lugar.
- **objetos** - Se utiliza como un directorio temporal donde se irán guardando los códigos objetos que se vayan creando como pasos previos a la creación del/los código(s) ejecutable(s) final(es).
- **misIncludes** - Este es un nombre arbitrario, pero se intenta separar los códigos fuentes del proyecto de aquellos códigos conseguidos de terceros (sea en código fuente, objeto y librerías, dll, etc.), y que son incluídos también como parte del proyecto.

Generalmente en la carpeta raíz del proyecto existe un Makefile que estará preparado para ejecutar todo lo necesario (incluso otros makefiles). Para este laboratorio omitiremos ésto y se crearán makefiles directamente en los directorios Sources y misIncludes según sea necesario.

Organización de proyectos utilizando el make.

La unión de los módulos para integrar un archivo ejecutable depende del tipo de archivo. Esta parte puede ser lo que más te confunda si no estás bien entrenado en las diferentes fases de compilación de un archivo fuente a un proyecto ejecutable; si tienes dudas pregunta, o lee la práctica relacionada con el complemento de C.

Como quiera, estos módulos pueden ser otros archivos de código fuente, código binario (archivo objeto o biblioteca), como los siguientes:

a) Si el módulo que se desea integrar es un archivo tipo texto, entonces se utiliza un comando del preprocesador para que realice la integración de los módulos antes de compilar. Ejemplo:

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

b) Si el módulo que se desea integrar es un archivo objeto, entonces la integración de los módulos se realiza en la última etapa de la compilación. Ejemplo:

```
gcc  -o  ejemplo  modulo1.o  modulo2.o  modulo3.o
```

```
indica ligar en un solo módulo objeto "ejemplo", otros tres objetos. El comando gcc (Gnu C Compiler) no solo compila C o C++; también puede integrar objetos previamente compilados
```

c) Si el módulo que se desea integrar es una biblioteca, entonces la integración de la biblioteca con los otros módulos también se realiza en la etapa final de la compilación, utilizando la opción *-lm <nombre de biblioteca>*. Ejemplo:

```
gcc  -o coseno -lm coseno.c
```

```
indica integrar la biblioteca "coseno.c" que parece incluír un solo módulo; en general, las bibliotecas incluyen varios módulos.
```

Cada uno de los casos se verán a continuación. Respecto a usarlos como prueba, puede presentar problemas el copiado de las líneas, así que si prefieres puedes descargar un archivo .tar que posee una copia sin errores en esta liga: Archivos de los 3 casos.

En general, las fases por las que pasa todo código en C hacia el archivo ejecutable son:

1. Pre-procesamiento - Solo aparece lenguaje C, un único archivo
2. Pre-compilador - Todo el lenguaje de alto nivel es traducido a lenguaje mnenomico de bajo nivel (Aún es texto).
3. Ensamblador - El lenguaje maquinal se convierte en código objeto (Propiamente un archivo que tiene instrucciones maquinales usualmente con la extensión .o).
4. Ligado - Se crea un único archivo ejecutable final.

5. Una programación avanzada, por ejemplo para el trabajo de un S.O. permite interrumpir el proceso de compilación de C en las primeras dos fases para modificarlo. La tercera fase es de mucha utilidad pues permite el compartir código objeto sin necesidad del código fuente. Pueden ver un ejemplo de estas fases en esta liga.

Caso 1) Integracion de dos archivos tipo texto (Método erróneo):

Para este caso considera la siguiente organización de directorios:

```
miProj
| - bin
| - doc
| - sources
|     | - proj.c
|     |- makefile
|- miIncludes
       |- aritReal.h
```

En el directorio *bin* quedará el programa integrado (códigos ejecutables o binarios).

En el directorio *sources* se encuentran el archivo fuente principal (**proj.c**) y el *Makefile*

El directorio *miIncludes* contiene todas los archivos locales al proyecto, que se pueden incluir al programa principal utilizando la directiva *#include* de lenguaje C.

A continuación se muestra el código fuente de los archivos: ***aritReal.h y proj.c***

```
/* archivo 'aritReal.h' es  almacenado en el directorio 'miIncludes'  */

/* función que suma dos números. */

float suma(float a, float b){
   return(a+b);
}

/* función que resta 'b' a 'a' */

float resta(float a, float b) {
  return(a-b);
}

/* función que multiplica dos números */

float multiplica(float a, float b){
  return(a*b);
}

/*función que divide 'a' por 'b'*/

float divide(float a, float b) {
  return(a/b);
}
```

Ahora, se muestra el archivo **proj.c** que invoca al archivo anterior:

```
/* archivo 'proj.c' almacenado en el directorio sources. */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "../miIncludes/aritReal.h" /* include definido por el usuario. */

int main(int argc, char **argv)
{
  float a,b,c,d;

  a= suma(3.5,3.4);
  b= resta(4.0,2.0);
  c= multiplica(3.0,5.0);
  d= divide(7.0,3.0);

  fprintf(stdout," a= %f\n b= %f \n c= %f \n d= %f\n", a, b, c, d);
}
```

Es muy común que el archivo M*akefile* se utilice para ver dependencias entre archivos y así solo se compilen aquellas partes del código que fueron afectadas. En este ejemplo, el archivo M*akefile(1)* se muestra a continuación:

```
CC= gcc   #/*string que se substituye mas adelante dondequiera que aparezca*/
ARGS=-O   #/*otro string. Indica: "optimiza el código" */

proj: proj.o   ../miIncludes/aritReal.h
<tab>$(CC) -o ../bin/proj $(ARGS) proj.o #borrar <tab> y oprimir la tecla Tab

proj.o: proj.c
<tab>$(CC)  $(ARGS)  -c  proj.c #borrar <tab> y oprimir la tecla Tab

clean:
<tab>rm   *.o  #borrar <tab> y oprimir la tecla Tab
```

Recordatorio: No olvidar el espacio TAB al inicio de las líneas siguientes a las etiquetas

En español, este archivo **Makefile nos dice que: "para obtener proj (primer objetivo), es necesario que existan proj.o y aritReal.h. Una vez que estos dos existen, se obtiene proj utilizando el compilador gcc, produciéndose como objeto proj (-o ..bin/proj) a partir de proj.o**. Ahora bien, para que pueda existir proj.o (segundo objetivo), se requiere que exista proj.c; una vez que existe proj.c, simplemente se compila. Si cambiara proj.c, al ejecutar el comando "make" se recompilaría. Si cambiara arithReal.h, pues también se recompilaría proj.c automáticamente, ya que el primer objetivo nos indica que arithReal.h es requisito para poder obtener proj y que éste depende de proj.o y que este último depende de proj.c que ya cambió pues incluye a arithReal.h". . . Bueno no se si lo prefieres así en español o mejor de manera más formal:

- CC y ARGS son constantes textuales que se sustituirán por sus valores donde quiera que aparezcan en el cuerpo del texto
- *proj* es un objetivo que tiene dependencia del código objeto, *proj.o*, y del archivo *../miIncludes/aritReal.h*. Es decir, si se ejecuta desde la línea de comandos "make proj" y se ha modificado alguno de estos dos archivos, entonces se ejecuta el comando que sigue, el cual obtiene el objetivo proj.
- El compilador nos dice que el código ejecutable se almacenará en el directorio *../bin/*.
- Nos dice que el archivo objeto proj.o tiene dependencia del código fuente proj.c. Es decir, que si se modifica el código fuente se obtendrá el código objeto de este fuente.

- Además, nos dice que el objetivo clean no tiene dependencias y por lo tanto, cada vez que ejecutemos esta opción ("make clean")se realizará el siguiente comando rm *.o, que

borrará del directorio todos los archivos objeto

Ahora, para obtener el código del proyecto se utiliza el comando **make** , como sigue:

```
user@localhost> cd miProj/sources
user@localhost> make
        cc -O -c proj.c
        cc -o ../bin/proj proj.o
user@localhost>
```

después de ejecutar el comando **make** exitosamente se puede observar que se creó el archivo *proj* en el directorio *../bin*, y el programa objeto *proj.o* esto es:

```
miProj
        | - bin
        |     |- proj
        | - doc
        | -sources
        |       | - proj.c
        |       | - proj.o
        |     |- Makefile
        | -miIncludes
            |funciones
```

El archivo objeto se puede borrar como sigue:

```
user@localhost> cd   miProj/sources
user@localhost> make  clean
```

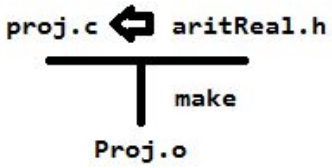
para ejecutar **proj**:

```
user@localhost> ../bin/proj
a= 6.900000
b= 2.000000
c= 15.000000
d= 2.333333
user@localhost>
```

Como el comando **make** tiene las dependencias entre archivos, entonces si intentamos volver a hacer el make, **no** ejecutará ninguna compilación, ya que no hemos modificado ni el archivo *aritReal*, ni el archivo *proj.c*.

> ***ADVERTENCIA**: Por propósitos didácticos, hemos usado un archivo denominado ArithReal.h que incluye el código fuente de varias funciones -suma, resta etc. Usualmente sin embargo, estos archivos ".h" sólo incluyen la primera línea de estas funciones, y el código de éstas proviene de otro lado. Los casos 2 y 3 ya se manejan de esta forma .*

En resumen, este caso compila directamente un archivo fuente el cual utiliza (de forma no usual) una librería que contiene código fuente a compilar junto con el archivo que lo incluye.



Caso 2) Integración de archivos textos y archivos objeto:

Este caso ya maneja un modo adecuado y correcto de utilizar las librerías. La función es similar pero existen más códigos fuentes que en el caso anterior y el make primero crea los códigos objetos y después los compila en un único archivo ejecutable.

Ejemplo:

Se utilizará el directorio *objetos* para almacenar el archivo fuente y el objeto de la biblioteca, y el directorio sources para almacenar el fuente y objeto del programa proj.c. Entonces, la organización de los directorios del proyecto es como sigue:

```
miProj
        | - bin
        | - doc
        | - objetos
        |       | - aritReal.c
        |      |- Makefile
        | - sources
        |       | - proj.c
        |      |- Makefile
    | - miIncludes
    |     | - aritReal.h
```

El archivo **M*akefile(2)*** en el directorio objetos contiene inicialmente el código fuente del archivo *aritReal.c*

El archivo M*akefile(2)* se muestra a continuación:

```
CC=gcc
aritReal.o: aritReal.c
<tab>$(CC) -c aritReal.c

clean:
<tab>rm  *.o
```

Nota: En el documento nos referimos a este archivo como Makefile(2) para diferenciarlo de los demás casos, pero en la carpeta debe llamarse “Makefile”.

para obtener el código objeto, se hace lo siguiente:

```
user@localhost> make  aritReal.o
        cc -c aritReal.c
```

Después de ejecutar el comando **make**, podemos ver que en el directorio se encuentra el código ejecutable del archivo *aritReal.c*, esto es:

```
user@localhost> cd  miProj/objetos
user@localhost>ls
aritReal.c    aritReal.o    Makefile
```

o en forma de árbol:

```
miProj
        | - bin
        | - doc
        | - objetos
        |       | - aritReal.c
        |       | - aritReal.o
        |      |- Makefile
        | - sources
        |       | - proj.c
        |      |- Makefile
```

Ahora que tenemos el código objeto, nos vamos al directorio *miProj/sources*. En este directorio se encuentran el código fuente *proj.c* y el archivo M*akefile* de nuestro proyecto.

A continuación se muestra el contenido del archivo M*akefile(3)*:

```
CC=gcc
ARGS=-O
proj:proj.o   ../objetos/aritReal.o
<tab>$(CC) -o ../bin/proj proj.o ../objetos/aritReal.o

proj.o: proj.c
```

```
<tab>$(CC) $(ARGS) -c proj.c

clean:
 <tab> rm   *.o
```

- Las dependencias de *proj* son sobre los código objeto *proj.o* y *../objetos/aritReal.o*.
- Al variar alguno de estos archivos se compila el archivo *proj.c* y se liga con los códigos objeto *proj.o aritReal.o*.

La ejecución del comando **make** sería como se muestra a continuación:

```
user@localhost> make proj
        cc  -O  -c  proj.c
        cc  -o   ../bin/proj proj.o   ../objetos/aritReal.o
user@localhost>
```

A continuación se muestra el archivo fuente *proj.c*

```
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "...objetos/aritReal.h"


int main(int argc, char **argv)
{
  float a,b,c,d;

  a= suma(3.5,3.4);
  b= resta(4.0,2.0);
  c= multiplica(3.0,5.0);
  d= divide(7.0,3.0);

  fprintf(stdout,"a= %f\n b= %f \n c= %f \n d= %f\n", a, b, c, d);
}
```

Aunque el encabezado aritReal.h está presente al igual que en el caso anterior, ahora solo posee los encabezados de las funciones como se muestra a continuación

```
/* archivo 'aritReal.h' es  almacenado en el directorio 'miIncludes'  */

/* función que suma dos números. */
float suma(float a, float b);

/* función que resta 'b' a 'a' */
float resta(float a, float b);

/* función que multiplica dos números */
float multiplica(float a, float b);

/*función que divide 'a' por 'b'*/
float divide(float a, float b);
```

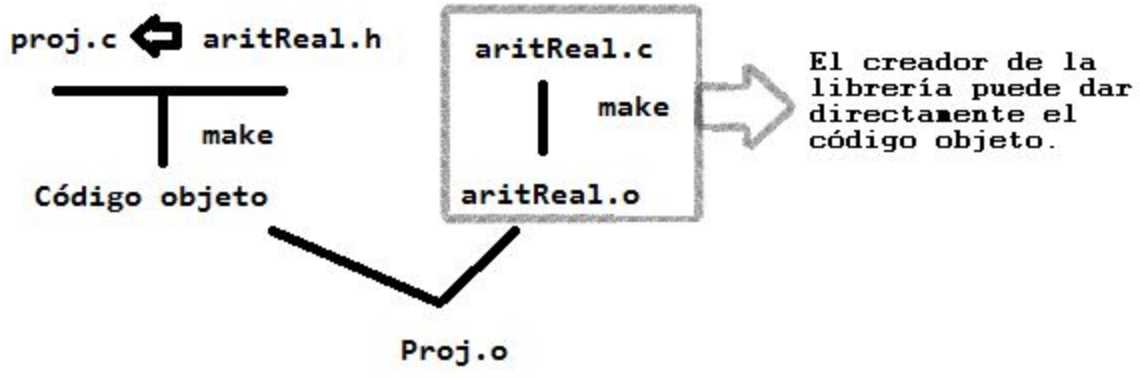
Aunque en el algunos casos se puede omitir hacer referencia a la librería se recomienda no hacerlo ya que pueden haber errores para el manejo de las funciones, lo cual se puede reflejar en valores inválidos como se muestra en la siguiente captura errónea:

```
user@localhost> proj
a= 1074528256.000000
b= 1074790400.000000
c= 1074266112.000000
d= 1075576832.000000
```

Posiblemente se esté preguntado ¿por qué utilizamos 2 archivos “Makefile”? Técnicamente no hay necesidad de hacerlo; un único archivo nos permitirá primero cumplir los objetivos de crear

los códigos objetos de la librería y después ligarlos al código objeto de nuestro código principal. La razón detrás de esta decisión es que usualmente cuando nos dan librerías, pueden no llegar a darnos los códigos fuentes sino solamente los códigos objeto; esto es muy común ya que la librería pudo haber sido desarrollada por otra empresa que desea es proteger el código fuente que ha desarrollado con sus propios recursos. Por eso hemos separado en dos partes el proceso de compilación: una parte que representa nuestro propio proyecto y en otra parte que representa la compilación de la empresa que nos proporciona su código.

En resumen, en este escenario generamos nuestro código objeto de nuestro proyecto y este se le añade el código objeto de la librería que en este caso es generado con otro archivo Makefile independiente.



Caso 3) Integración de archivos texto y bibliotecas:

Este es un caso común y en otros lenguajes es como si consiguieran el archivo .dll o los .class listos para integrarse al código de su proyecto. En este caso muy en particular utilizaremos una biblioteca (archivo binario) generada de forma estática y nuestro código fuente.

Para este tercer caso, se maneja una biblioteca la cual puede estar constituida de la combinación de todos los códigos objetos de una compañía particular en un único archivo; dicho archivo es la biblioteca que nos proporciona esa compañía. El detalle viene en que los S.O. modernos se manejan dos tipos de librerías: dinámicas (usualmente compartidas) y estáticas. Las primeras tienen como objetivo ahorrar espacio de memoria en aquellas bibliotecas que son utilizadas por más de un programa; por lo general la biblioteca se anexa al programa en el momento de la ejecución del mismo.

En el caso de las librerías estáticas, las librerías son anexadas al programa que las utiliza al final del proceso de compilación. Esta fase final se denomina “ligado” o “linking” en inglés. En este escenario, más de un programa puede utilizar la misma librería pero estarán duplicadas en cada programa.

Para objetivos de esta práctica se manejarán exclusivamente librerías estáticas. Para mayor información de cómo crear una biblioteca dinámica ver el Anexo1 al final del documento.

Ejemplo:

Crear un biblioteca que contenga el código almacenado en el archivo *aritReal.o*.
La organización del proyecto es similar a la del ejemplo anterior:

```
miProj
        | - bin
        | - doc
        | - biblioteca
```

```
|     | - aritReal.c
|     |- Makefile
| - sources
|     | - proj.c
|     |- Makefile
| - miIncludes
|     | - aritReal.h
```

El archivo M*akefile(4)* del directorio *miProj/biblioteca* contiene la siguiente información:

```
modules= aritReal.o

all:libreria

aritReal.o: *.c
<tab>gcc -c -Wall *.c

libreria: $(modules)
<tab>ar -cvq libMia.a *.o
clean:
 <tab>rm   *.o
 <tab>rm   libMia*
```

Observe que en esta ocasión se están utilizando dos programas distintos: **gcc** y **ar**; el primero es el compilador mientras el segundo se utiliza para crear una biblioteca que contenga el resultado de las compilaciones. Se ha añadido el objetivo "all" para garantizar que el uso del comando make genere los códigos objetos y la creación de la librería (en realidad, inicia la creación de librería pero para lograrlo debe cumplir primero con el objetivo aritReal.o).

- La opción “-Wall” dentro del objetivo "aritReal.o" indica al compilador que se compilen todos los codigos fuente obteniéndose los códigos objeto correspondientes. En este caso hay un solo código fuente a compilarse pero en el caso general podría haber varios.
- El objetivo librería se encarga de colocar todos los códigos objeto dentro de la que será la librería estática libMia.a

Para crear la biblioteca *libMia.a*:

```
user@localhost> ls
aritReal.c    Makefile
user@localhost> make
gcc -c -Wall *.c
ar -cvq libMia.a *.o
user@localhost> ls

aritReal.c  aritReal.o  libMia.a  Makefile

user@localhost>
```

Ahora, que ya se creó la biblioteca nos cambiamos al directorio *miProj/sources*. El archivo *proj.c* está desplegado a continuación:

```
/* proj.c */
#include <stdio.h>
#include <stdlib.h>
#include <string.h>
#include "../miIncludes/aritReal.h"

int main()
{
```

```
   float a,b,c,d;

   a= suma(3.5,3.4);
   b= resta(4.0,2.0);
   c= multiplica(3.0,5.0);
   d= divide(7.0,3.0);
   fprintf(stdout,"a= %f\n b= %f \n c= %f\n d=%f\n", a, b, c, d);
 }
```

El archivo M*akefile(5)* se muestra a continuación:

```
DIR=../biblioteca
CC=gcc

proj: proj.o
	$(CC) -o ../bin/proj  proj.o   -L$(DIR)  -lMia

proj.o: proj.c

clean:
	rm  *.o
```

- La opción *-lMia* indica que se va a utilizar una biblioteca extra diferente a la biblioteca estándar y lleva el nombre *libMia.a*
- La opción *-L* sirve para indicar un directorio donde puede encontrar la biblioteca.
- El objetivo proj.o solo tiene como dependencia proj.c. Como puede observar se genera el código objeto sin necesidad de una linea de gcc. Esto es automático cuando el objetivo es un objeto y la dependencia un solo fuente del mismo nombre que el objetivo.

Para compilar el programa se hace lo siguiente:

```
user@localhost> cd miProj/sources
user@localhost> make

gcc    -c -o proj.o proj.c
gcc -o ../bin/proj  proj.o   -L../objetos  -lMia
user@localhost>
```

El archivo cabecera *aritReal.h deja de ser una opción y* se muestra a continuación:

```
 /* archivo 'aritReal.h'  es  almacenado en
    el directorio 'miIncludes'  */

 /** funcion que suma 'a' y 'b' */
 float suma(float a, float b);

 /** funcion que le resta 'b' a 'a' */
 float resta(float a, float b);

 /**función que multiplica 'a' y 'b' */
 float multiplica(float a, float b);

 /** función que divide 'a' por 'b' */
 float divide(float a, float b);
```

La organización de proyecto se vería así:

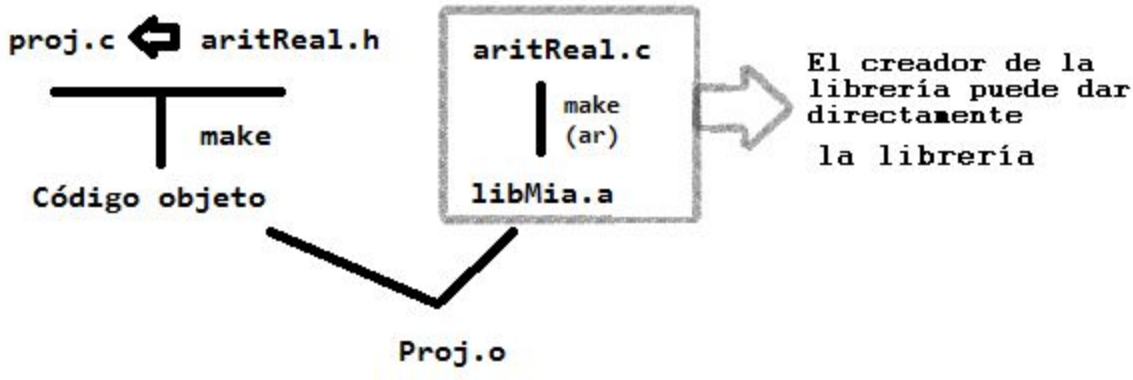
```
miProj
       | - bin
       | - doc
       | - objetos
       |      | - aritReal.c
       |      | - aritReal.o
       |      | - libMia.a
       |      |- Makefile
```

```
| - sources
|       | - proj.c
|       |- Makefile
| -miIncludes
        |- aritReal.h
```

A continuación se muestra la codificación del programa principal, proj.c:

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

#include "../miIncludes/aritReal.h"

int main()
{
  float a,b,c,d;

  a= suma(3.5,3.4);
  b= resta(4.0,2.0);
  c= multiplica(3.0,5.0);
  d= divide(7.0,3.0);

  fprintf(stdout,"a= %f\n b= %f \n c= %f \n d=%f %f\n", a, b, c, d);
}
```

En resumen este caso es similar al caso 2, se compila el código fuente de nuestro proyecto y a este se le vincula con la librería previamente creada mediante otro makefile (por la forma en que representamos de donde salió la librería). En este escenario el encabezado (aritReal.h) es obligatorio.



= = = = Laboratorio = = = =

- Actividades a realizar en esta práctica se encuentran descritas en este documento, sus respuestas deben registrarse en :

    - **Material de Apoyo**: Lab-06: Make
    - **Enlace al formulario:** Lab 06

- Lee el manual de make
- Desarrolla el programa *numerosUno.c* que calcule el cubo, la cuarta y la quinta potencia de un número que se pasa por la línea de comando. Los valores se calcularán usando las funciones especificadas en la Tabla 1:

Nombre de la función	Nombre del archivo que lo contiene	¿Qué hace?
cubo	cubo.c	Función que recibe como entrada un número y regresa su cubo.
cuarta	cuarta.c	Función que recibe como entrada un número y regresa su cuarta.
quinta	quinta.c	Función que recibe como entrada un número y regresa el número elevado a la quinta.

Tabla 1. Funciones matemáticas.

Ejercicio 1

Los resultados se deberán imprimir en pantalla **y guardar en el archivo *salidaUno.txt***. Una vez que tengas el los archivos de código: *numerosUno.c,, cubo.c, cuarta.c, quinta.c (nota que son cuatro archivos separados, no uno solo con todas estas funciones),* c**rea un archivo único M*akefile* que te permita compilar todos los archivos**.

Primero define tres reglas que obtengan el código objeto de los archivos *cuadrado.c, cubo.c* y *cuarta.c*; y luego define otra regla para crear el programa *numerosUno* a partir de los archivos *numerosUno.c, cuadrado.o, cubo.o* y *cuarta.o*. También define una regla *clean* para borrar los archivos objeto después de la compilación.

Ejercicio 2

Desarrolla la biblioteca *potencias*, que contiene las funciones especificadas en la Tabla 1 (un único archivo). A partir de el programa anterior, desarrolla el programa *numerosDos.c,* que realiza las mismas operaciones aritméticas, pero que utiliza la biblioteca *potencias* en lugar de los archivo objeto como fuente de código para las funciones aritméticas.

Lecturas sugeridas:

La siguiente liga posee ejemplos de Make: http://mrbook.org/tutorials/make/

Anexo 1 - Bibliotecas compartidas

Creación

Para crear una biblioteca dinámica se necesita tener disponibles los archivos objetos generados con las banderas -fPIC o fpic la cual habilita el posicionamiento del código independiente. La bandera -Wl sirve para mandarle opciones al programa encargado de hacer las ligas (linker) y no debe tener espacios sin escapar. Para generar la biblioteca se usa el siguiente comando.

```
gcc -shared -Wl,-soname,your_soname \ -o library_name file_list library_list
```

donde el "your_soname" es el nombre de la biblioteca compartida que comienza con "lib", el nombre de la biblioteca y seguida por la extensión ".so"

Compilar programas

Para compilar programas que hagan uso de una biblioteca compartida es necesario utilizar la bandera -L para especificar el lugar donde se encuentra dicha biblioteca.

```
gcc -L /path/a/bilbioteca/ -libBiblioteca código.c
```

Ejecutar programas usando bibliotecas dinámicas

Si no se instaló la biblioteca en un lugar estándar se puede utilizar la variable de entorno LD_LIBRARY_PATH para especificar dónde se puede encontrar la biblioteca que se necesita para ejecutar el programa.

```
LD_LIBRARY_PATH=/path/a/mi/biblioteca/;$LD_LIBRARY_PATH mi_programa
```

Referencias

http://www.dwheeler.com/program-library/Program-Library-HOWTO/x36.html