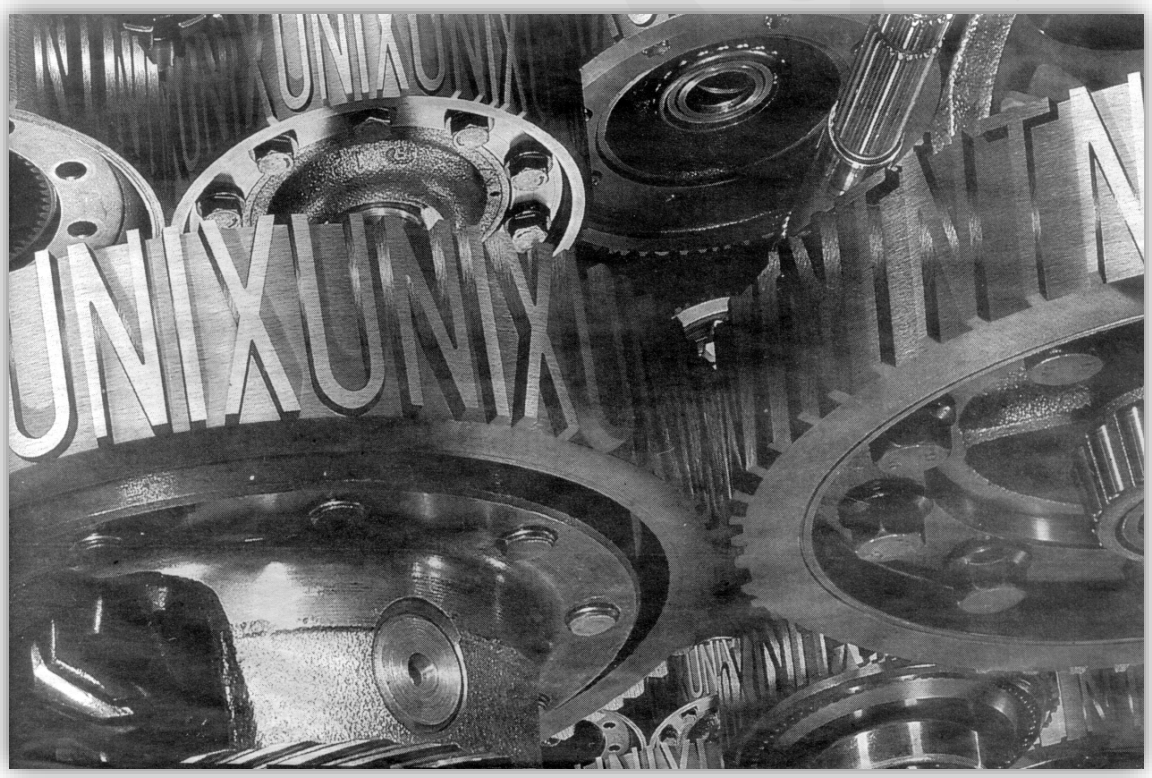
Universidad de Cantabria
Grado de Ingeniería Informática

Sistemas Operativos

Práctica 1 Parte 3



Rafael Menéndez de Llano Rozas

Guion

1

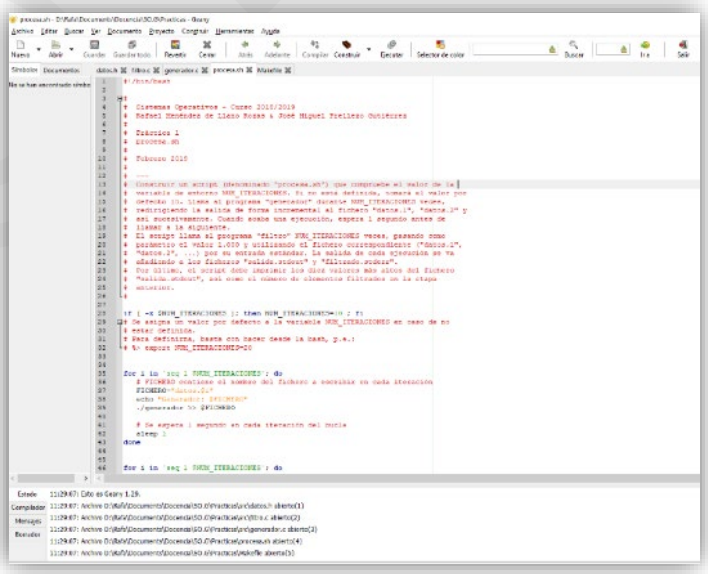
Uso de la Shell y del entorno de desarrollo del sistema

En este apartado explicamos y usamos las herramientas de desarrollo de programas (entorno de programación).

Entorno de desarrollo

Un entorno de desarrollo se compone fundamentalmente de tres tipos de herramientas:

1. Editor de texto para la edición de código fuente. Puede ser:
   - De texto. De pantalla como `vi` (`vim`), `nano` o de línea como `ex` o `ed`.
   - Gráfico. Dependerá del entorno de ventanas en uso o podrá ser independiente. Por ejemplo, `kedit` o `gedit` y `sublime`, `emacs` o `nedit`. Las propiedades principales son: Marcado sintáctico (muy útil para programar, también el `vim` lo tiene si está configurado) y compilación integrada.
   - Una herramienta muy interesante, multilenguaje y multiplataforma para desarrollar código es Geany (www.geany.org).



2. Compilador y afines: Herramienta que permite convertir secuencias de control en alto nivel a lenguaje máquina.
   - `gcc` en modo línea de comando.
   - `makefile` para compilación automatizada.

3. Depurador. Herramienta que permite comprobar en tiempo de ejecución por qué un programa no funciona. Puede ser en:
   - Modo línea de comando: `gdb`.
   - Modo gráfico: `ddd`.

Incluso existen otros tipos de herramientas que permiten la colaboración entre usuarios para la realización de un proyecto como por ejemplo el control de versiones –`git`– y repositorios que serán tratados en otras asignaturas.

También podemos disponer de entornos integrados de desarrollo (IDE) que agrupan todas estas herramientas en una única y pueden estar dedicados a varios lenguajes de programación como Eclipse, ActiveState Komodo, IntelliJ IDEA, Oracle JDeveloper, NetBeans, Codenvy o Microsoft Visual Studio. Hay otros, que estando basados en un lenguaje concreto, soportan varios a través de plug-in como GNU Emacs, basado en C, también para ADA, Lisp, PHP, Perl…; o Eclipse o NetBeans, basados en Java, o MonoDevelop, basados en C# sirven para otros tantos. También existen IDE que vienen con el propio gestor de ventanas como Kdevelop, Builder, Anjuta…

En nuestro caso, vamos a trabajar con las herramientas de forma independiente. Haciendo una analogía, vamos a aprender a sumar antes de usar la calculadora.

De los editores no hay mucho que decir salvo que ya se ha utilizado el `vi - vim` para la escritura de macros y se han visto sus carencias a pesar de su universalidad. Por eso, a partir de aquí, se podrá usar cualquiera de los editores del sistema. Pasemos por tanto a la siguiente herramienta.

Compilador

El compilador de C que vamos a utilizar en el laboratorio es compilador de C de GNU, el `gcc` (g++). Este compilador realiza automáticamente toda la cadena de operaciones para producir un fichero ejecutable, que se llama por defecto "`a.out`" siguiendo un antiguo formato de ejecutables, es decir:

1. Llama al preprocesador. `cpp` y produce código fuente C puro.
2. Compila el programa y produce un fichero en ensamblador.
3. Ensambla el programa, `as` es el ensamblador, y produce un código objeto.
4. Enlaza (*linka*) el objeto, `ld` es el enlazador (*linker*), y produce el ejecutable.

El compilador siempre espera que se le dé un nombre de fichero que contenga un programa fuente de lenguaje C, para distinguir estos ficheros de código fuente del resto de ficheros del sistema, es obligatorio que todos terminen en ".c". Si no hacemos esto, el compilador nos responderá con: "`nombre: file not recognized: No se reconoce el formato del fichero`". Otras terminaciones comunes son:

- .C, .cc, y .cxx para ficheros en c++.
- .h, .i para ficheros del preprocesador (ficheros de cabecera .h o *includes .i*).
- .s, .S para ficheros en ensamblador.
- .o para ficheros objeto (sin enlazar).

Como cualquier otro comando del sistema, el formato para ejecutarlo será nombre [-opciones] y argumentos. La forma más sencilla de ejecutarlo para producir un fichero ejecutable con nombre `a.out` sería la siguiente:

```
gcc programa.c
```

Como hemos dicho, el compilador admite opciones, estas usualmente se pondrán delante del nombre del fichero fuente y pero a diferencia de otros comandos deberán estar separadas, por ejemplo, no es lo mismo poner `gcc -dr` que `gcc -d -r`. Las opciones están divididas en varios grupos: globales, del lenguaje, de alerta, de depuración, de optimización, de preprocesado, de ensamblador o linkador, de directorios y de dependencias del hardware.

Las opciones más habituales (las puedes encontrar con `man gcc`) son:

- **Globales**:
  - –c Compila pero no llama al linker, generando un objeto con extensión ".o".
  - -o nombre Cambia el nombre del fichero ejecutable (por defecto `a.out`) al expresado en la opción nombre.
- **Del lenguaje**:
  - -ansi Compila siguiendo las reglas del ANSI C.
  - -std=std Compila siguiendo el estándar std, por ejemplo, c99.
  - -E Ejecuta solo el preprocesador.
  - –D Cambia el valor de una constante del preprocesador desde la línea de comando al compilar. La constante no está definida.
- **Del linker**:
  - -llibreria Incluye la librería libreria. Se pone al final de la línea.
  - –static Para *linkar* de forma estática.
- **De directorios**:
  - -Ldirectorio Añade el directorio directorio a los directorios donde se buscará las librerías, por defecto `/usr/lib`.
  - -Idirectorio Añade el directorio a la lista de directorios donde están los ficheros de cabecera .h, por defecto `/usr/include`.
- **De alerta**:
  - -w Inhibe los mensajes de *warning*. Los mensajes de *warning* indicarán algo a tener en cuenta, pero que no es un error y se produce el ejecutable.
  - -Wall Pone todos los mensajes de *warning*. Recomendable al empezar.
- **De depuración**:
  - -g Introduce información adicional en el ejecutable para poder utilizar alguno de los depuradores (*debuggers*) del sistema (`gdb`).
- **De optimización:**
  - -O Se ponen los niveles de forma numérica: -O0 Sin optimizar, -O1 Optimización razonable, -O2 mejor, -Os mejor sin incrementar tamaño, -O3 agresiva.

Una vez que se ha compilado un programa con la opción -g, puede ser ejecutado con el *debugger* del sistema (gdb –línea– o ddd –gráfico–). Esto permitirá ejecutarlo de forma controlada (paso a paso, línea a línea, hasta un punto de ruptura…), ver los valores de las variables, poner guardianes para cambio de variables, etc. Para más información se puede utilizar el comando *help* del mismo. También se puede utilizar el depurador para obtener información de los ficheros *cores* del sistema. Cuando un programa se aborta en ejecución en algunos casos se produce un fichero de nombre `core` que puede ser analizado con `gdb core`. Se puede forzar su aparición con `ulimit -c unlimited`.

El compilador no solo toma como argumentos ficheros fuente, si hemos hecho el programa en módulos (módulos objeto compilados, ficheros cabecera, ficheros del preprocesador…). Un ejemplo de compilación de este tipo sería:

```
gcc modulo1.c modulo2.c prepro.i programa.c objeto.o -o ejecutable -lm
```

1. Vete a la asignatura virtual y bájate el fichero comprimido de ejemplos para el sistema de desarrollo. Selecciona el fichero `min.c`. Ábrelo con el editor de textos y revísalo. Después compílalo con el gcc. Usa a continuación varias opciones para ver el resultado como `-ansi, -std=c99` o `-Wall`.

2. En el fichero `min.c` hay una constante que se llama MAXIMO, debes comentar la línea de definición y dar valor desde fuera con la opción –D `gcc -DMAXIMO=20 …`

3. Ejecuta el programa. Verás que te pide una serie de datos. Bájate el fichero `datos` y utilízalo con la redirección de entrada. Usa también la redirección de salida. Introduce un error en el programa e indica hacia que canal va la salida del compilador. Usa la opción `-o` para generar un programa ejecutable distinto de `a.out`. Con la opción `-g` indica cómo cambia el tamaño del ejecutable. Usa el comando `nm` para ver los símbolos del programa, con y sin `-g`. Con la opción `-static` haz que el programa se enlace de forma estática (no dinámica por defecto) y mira la diferencia de tamaño.

4. Selecciona los ficheros `main.c, cap.c y cal.c`. Ábrelos con el editor de textos y revísalos. Crea una línea de compilación para producir un ejecutable con estos ficheros. Usa la opción `-c` con `cal.c` para producir un fichero objeto y después repite la línea de compilación con este objeto. Revisa el fichero `partes.c` e indica como lo compilarías.

5. Selecciona el fichero `cal2.c`. Ábrelo con el editor de textos y revísalo. Indica la diferencia con el anterior. Crea una línea de compilación para producir un ejecutable (usa la librería matemática, revisa también las pruebas comentadas en `main.c`). Indica que es `math.h` y la opción `-lm`. ¿Dónde localizarías la librería? Usa la opción `-c` con `cal2.c` para producir un fichero objeto y después repite la línea de compilación con este objeto.

Cuando el programa está compuesto de varios ficheros fuente es muy engorroso y poco eficiente usar la línea de comando. Por ello se utilizan otras herramientas más sofisticadas como el `make`.

La herramienta *make*

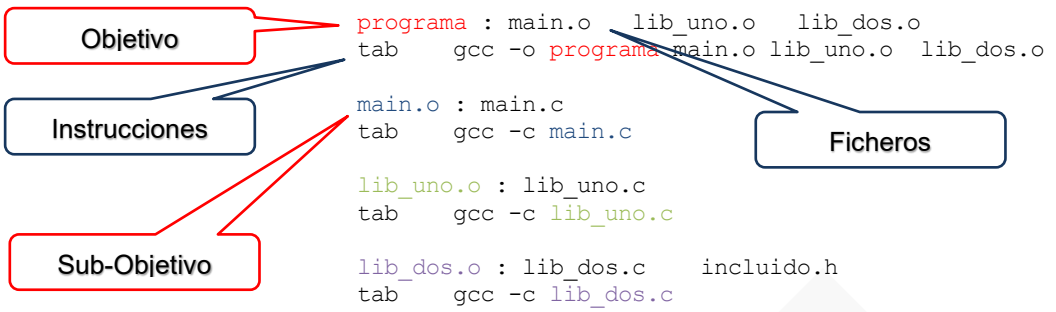
Otra herramienta interesante cuando se hacen grandes programas que están compuestos de muchos ficheros es `make`. Esta se utiliza con un fichero de comandos donde decimos de qué ficheros está compuesto nuestro programa, qué librerías utiliza y cómo se pueden obtener los ejecutables. Hay que recordar que un programa en C puede estar compuesto de varios ficheros, uno de ellos tendrá la función `main()` y los otros serán otras funciones ya compiladas con la opción -c (sin llamar al *linker*), código fuente, librerías o ficheros de cabecera `.h`.

La ventaja de utilizar el `make`, es que nos compilará solo los ficheros que sean necesarios según la fecha y hora del último cambio y que no tendremos que poner una complicada línea de comando para realizar la compilación, solo `make`, el cual leerá el fichero de órdenes `makefile` (es el nombre por defecto), donde encontrará cómo hacer la compilación (`make` podría funcionar sin fichero para una compilación realmente simple, por ejemplo, si tengo un fichero `hola.c` y quiero un ejecutable hola, bastaría con hacer `make hola`).

El fichero de órdenes se compondrá de una serie de *targets* (objetivos) o reglas (*rules*) a realizar (dependencias) y de la descripción de cómo realizarlas (indicación para la máquina), las líneas de realización siempre empiezan con el carácter `tabulador` (ojo que nadie ponga tab), pudiendo haber varias para un solo objetivo.

En el siguiente ejemplo el fichero `makefile` estará compuesto de cuatro objetivos: `programa, main.o, lib_uno.o, y lib_dos.o`. El primer objetivo es el fichero ejecutable `programa` cuyas dependencias aparecen a continuación de los ":". En la siguiente línea (empieza por el carácter tab)

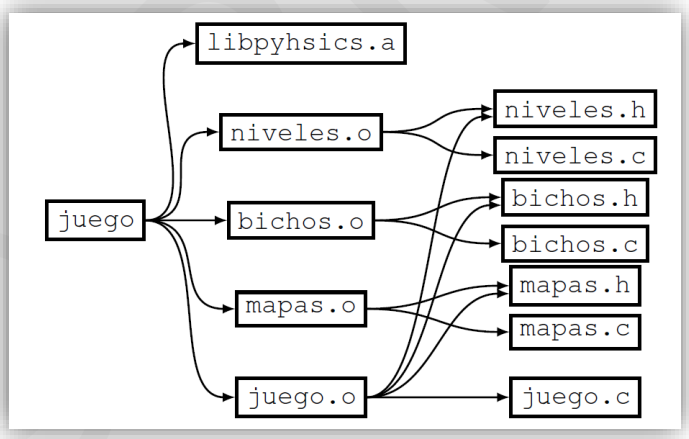
se dice cómo se puede obtener, en este caso compilando e incluyendo los ficheros objeto. A continuación, se indica cómo obtener estos objetivos secundarios, en este caso compilando con la opción -c para producir ficheros objeto. Se observa que el objetivo lib_dos.o tiene dos dependencias: lib_dos.c e incluido.h, ya que lib_dos.c tiene una instrucción de tipo #include <incluido.h>.



Como hemos dicho, si el fichero se llama `makefile` bastará con escribir `make` para hacer la compilación, si lo hemos llamado de otra manera deberemos usar la opción `-f`. También se puede poner como argumento el objetivo que queremos alcanzar, se toma por defecto el primero.

```
make -f ficherodemake  objetivo
```

6. Dada la figura posterior, escribe una línea de comando para realizar la compilación del programa juego. Después escribe un fichero *make* atendiendo a las dependencias descritas.



Al igual que en las macros, en los ficheros `make` también se pueden usar variables, incluso emplear las variables de entorno definidas en la Shell. Hay bastantes definidas por defecto que se pueden redefinir, por ejemplo: `CC`, `CFLAGS` u `OBJECTS` (es distinto usar := -no se verá aquí-):

```
CC=gcc
CFLAGS=-Wall -g -O2
OBJECTS = main.o lib_uno.o lib_dos.o
programa : $(OBJECTS)
	$(CC) -o programa $(OBJECTS)
main.o : main.c
	$(CC) -c main.c
lib_uno.o : lib_uno.c
	$(CC) -c lib_uno.c
lib_dos.o : lib_dos.c    incluido.c
	$(CC) -c lib_dos.c
```

O usar las variables de entorno:

```
SRC = $(HOME)/src
juego:
tab   gcc $(SRC)/*.c –o juego
```

Al estilo de las variables de la *bash* existen comodines automáticos (*wildcards*) en `make` que también se pueden usar (hay muchas más) y se pueden pintar con `@echo`:

```
$@: Se sustituye por el nombre del objetivo de la presente regla.
$<: Se sustituye por la primera dependencia de la presente regla.
$^: Se sustituye por una lista de dependencias de la presente regla.
$?: Se sustituye por una lista de dependencias de recientes.
```

Así, se podría convertir una regla como esta:

```
CC=gcc
CFLAGS=-Wall -g

hola: hola.o auxhola.o
tab    @echo "voy a compilar" $^
tab    $(CC) -o hola hola.o auxhola.o

#convertida con variables automaticas
hola: hola.o auxhola.o
tab    $(CC) -o $@ $^
```

También hay una serie de reglas (*rules*) por defecto de solo una línea, así se podría sustituir el tercer objetivo por esto (se usan cuando las dependencias son obvias):

```
lib_uno.o :
```

7. Haz un fichero *make* para la compilación automatizada de los tres ficheros `main.c`, `cap.c` y `cal.c`. Pruébalo. Vuelve a ejecutarlo ¿qué ocurre? Cambia la fecha de algunos de los ficheros fuente, ejecútalo y observa que ocurre. Cambia tu fichero `makefile` (haz una copia) para usar en este caso `cal2.c` Cópialo de nuevo y produce una versión para depuración. Copia de nuevo el fichero y haz una versión con las reglas por defecto y variables automáticas (*wildcards*).

Por último, existen una serie de reglas virtuales que complementan a las reglas normales que sirvan para realizar una determinada acción dentro de nuestro proyecto. El ejemplo más típico de este tipo de reglas es la regla *clean*, utilizada para "limpiar" de ficheros objeto los directorios que haga falta, con el propósito de rehacer todo la próxima vez que se llame a *make* o para borrar ficheros no deseados como los *core*. Solo se ejecutará si hacemos `make clean`.

```
clean:    # para cores y objetos
tab     rm core
tab     rm –f juego *.o
```

Otro objetivo virtual común se usa para imprimir ficheros:

```
print:
tab     lp *.c
```

Como norma general, se puede poner un objetivo que sea una combinación de comandos, que solo se ejecutarán cuando se haga explícitamente `make objetivo` (si no se pone nada se ejecutan las reglas normales). Si el objetivo coincide con algún fichero se puede usar previamente la tarjeta `PHONY`:

```
.PHONY: clean
```

8. Añade al fichero `make` una regla regla virtual para borrar los ficheros innecesarios, añadir los ficheros que quieres distribuir a un fichero `tar` y después comprimirlo `gzip` (ver el último apartado de empaquetado y compresión).

Programación separada (modular) - solo información

Hasta ahora hemos visto que todos nuestros programas estaban en un fichero que se edita, compila y ejecuta. Pero normalmente, en programas grandes, esto no se hace así, sino que se construye el programa de forma modular en varios ficheros, aplicándose el principio de *divide y vencerás*, ya que los módulos del programa serán más fáciles de entender y depurar (algo parecido a la división de un programa en funciones, pero a otro nivel más elevado). Con esta forma de trabajar conseguimos algunas ventajas:

1. Obviamente los módulos tienen una extensión menor que el programa completo. Por lo tanto, éstos serán más fáciles de analizar.
2. Cada módulo se puede compilar por separado (más rápido) y será más fácil de depurar, ya que no se tendrán que tener en cuenta influencias externas.
3. La división del trabajo entre varios programadores es más sencilla y limpia de realizar.

Esto conlleva que sea conveniente aplicar ciertos criterios a la hora de construir ese programa utilizando la estructura en árbol de directorios y ficheros:

1. Se puede utilizar un directorio (en vez de un fichero como antes) para contener los ficheros de los que va a estar constituido el programa.

2. Dentro de ese directorio general se puede crear varios subdirectorios donde se sepa que se va a encontrar lo que estamos buscando como, por ejemplo:
    2.1. Un directorio para los ficheros fuente.
    2.2. Un directorio para los ficheros de cabecera del preprocesado (normalmente "include"). Ver punto 4. Los del sistema están en el directorio "`/usr/include`".
    2.3. Un directorio de librerías (normalmente "lib").
    2.4. Un directorio de ejecutables (normalmente "bin").
    2.5. Un directorio de documentación (normalmente "doc").

3. Los módulos tienen que ser construidos (división del programa) teniendo en cuenta principios semánticos, ofreciendo servicios a otros módulos externos (cajas negras), de tal manera que se garantice el perfecto funcionamiento de los mismos de forma aislada. También se deberá tener en cuenta que (algunos inconvenientes del código dependiente se pueden solventar con la compilación condicionada que nos proporciona el preprocesador):
    3.1. Hay partes dependientes del hardware que deben ser señaladas como tal. De hecho, por definición no son transportables a otros sistemas y deben estar separadas del resto del programa.
    3.2. Otras dependerán de algo específico como llamadas a un sistema operativo concreto y deberán ser tratadas de la misma forma.

4. Los ficheros de cabecera "*.h" están destinados normalmente a contener las definiciones comunes a varios módulos. En ellos suelen aparecer distintos tipos de información:
    4.1. Definición de constantes.
    4.2. Definición de tipos de datos.
    4.3. Definición de prototipos de funciones.

    Suele ser una mala práctica de programación incluir las propias definiciones (reservas de espacio) de variables.

Gestión de librerías

Anteriormente se ha comentado que podemos incluir en nuestro programa código objeto realizado en otros lenguajes o en el mismo C, así la línea de compilación que veíamos en el anterior apartado podría complicarse:

```
gcc main.c modulo1.c modulo2.c prepro.i programa.c objeto.o -o eje -lm
```

donde hemos incluido dos módulos de código fuente C, un fichero de preprocesado, el programa principal fuente, un código objeto (no necesariamente C) y una librería, en este caso la librería matemática (para ello deberemos haber usado en alguna parte un `#include de math.h`).

Una pregunta que podemos hacernos es por qué hemos incluido esa librería. La respuesta es porque con el fichero de cabecera `math.h` solo hemos incluido definiciones de constantes, tipos de datos y prototipos de funciones, pero no el cuerpo compilado de estas funciones, que está contenido precisamente en esa librería (`libm.a libm.so`), ya que estas funciones no son de uso general. Ocurre lo contrario con las librerías de manejo de la entrada/salida y funciones comunes, que sí que se incluyen por defecto al realizar los ejecutables, estas librerías son `crt0.o` y `libc.a` (podemos considerar una librería como un conjunto de ficheros que contienen código objeto, las propiedades de esos ficheros son asimiladas en la propia librería que los mantiene a través de un registro índice).

De la misma manera que existen librería predefinidas, podemos crear las nuestras. Para ello existe el comando `ar`, que nos permite añadir módulos, modificarlas o eliminarlas (quitar módulos). Como cualquier comando, la sintaxis del mismo incluye opciones y argumentos:

```
ar -[opciones] [módulos] librería [ficheros]
```

Las opciones más habituales son:

Opción	Significado
d	Borrar módulos a través de los ficheros indicados
m	Cambia de orden (mueve) un módulo en la librería
p	Pinta en pantalla un módulo a través de su fichero
q	Añade de forma rápida módulos (sin registro índice) al final
r	Reemplaza módulos a través de su fichero
t	Muestra el contenido de la librería
x	Extrae módulos a través de su fichero
o	Preserva la fecha original del módulo en la extracción
s	Crea o actualiza el registro índice
u	Reemplaza teniendo en cuenta la fecha
Modificador	
a	Lo coloca detrás de un módulo existente
b, i	Añade delante de un módulo existente
c	Crea una librería
v	Modo "verbose"

De esta manera si tenemos una librería que se llama `libre.a` y tres módulos `mod1.o`, `mod2.o` y `mod3.o` podemos hacer:

Ejemplo	Acción
ar c libre.a	Crea la librería
ar r libre.a mod1.o	Añade el módulo y crea la librería si no existe
ar tv libre.a	Muestra el contenido de la librería
ar q libre.a mod2.o mod3.o	Coloca al final de forma rápida el módulo
ar s libre.a	Actualiza el registro índice
ar x libre.a mod3.o	Extrae el tercer módulo

A la hora de usar la librería creada tenemos que tener en cuenta las siguientes reglas:

1. Las librerías se buscarán en los directorios por defecto que son `lib, lib64` y `/usr/lib`. Si no ponemos nuestra librería ahí (no nos dejaran), tendremos que utilizar la opción del compilador -L para indicarlo.
2. Lo mismo tendremos que hacer con los ficheros incluidos de cabecera, en este caso la opción es -I. El directorio por defecto es `/usr/include`.
3. Todas las librerías que creemos empezaran con la palabra `lib` a la que seguirá el nombre propio de la librería (en el caso de las matemáticas `m` y en el anterior `re`).
4. Al compilar tendremos que invocar al enlazador con la opción -l para que incluya la librería creada (en el ejemplo `-lre`).

Existe un comando relacionado con las librerías, comando `nm`, para ver el contenido de sus módulos:

```
nm -[opciones] [ficheros]
```

donde el fichero puede ser un módulo o una librería, en este último caso se puede usar la opción -s para ver el índice.

Depuración (modo línea)

Una vez compilado el programa con éxito, tendremos que ejecutarlo. Lo normal es que existan errores de ejecución. Para depurarlos se pueden realizar impresiones de variables por nuestra cuenta (ojo, recuerda que el `printf` sin `\n` no imprimirá un mensaje inmediatamente por la salida estándar) o preferentemente utilizar un depurador si el programa es secuencial (si es concurrente la depuración será más compleja).

Dentro de las herramientas fundamentales de desarrollo incluidas en el `GNU toolchain` (ya habéis usado entornos integrados y los seguiréis usando en un futuro, pero conviene que sepáis las herramientas básicas) hay una aplicación de depuración de línea de comandos que es el `gdb`.:

- GNU Compiler Collection (GCC): compiladores para varios lenguajes.
- GNU Binutils: enlazador, ensamblador y otras herramientas.
- GNU C Library (glibc): librería de C, incluyendo cabeceras, librerías y el cargador dinámico.
- **GNU Debugger (GDB): un depurador interactivo.**
- GNU make: automatización de la estructura y de la compilación.

Antes de poder usarlo, debemos producir un ejecutable que contenga información extra del código fuente de lenguaje C y para ello debemos compilar con la opción –g, por ejemplo:

```
gcc -Wall -ansi -g prog.c -o prog
```

Después bastará con abrir el `gdb` y poner como argumento el programa generado o no poner nada y abrirlo con el comando interno `file` (tienes una referencia de los comandos en http://users.ece.utexas.edu/~adnan/gdb-refcard.pdf). Los comandos internos funcionan con las mismas ayudas que la Shell (cursores, tab…), pero en este caso, la ayuda viene del comando `help` y muchos admiten formas abreviadas con una letra, por ejemplo, `q` por `quit`. Si no tuviéramos errores, bastaría con utilizar `run (r)` y el programa se ejecutaría normalmente, pero obviamente, si estamos usando `gdb` es que no es así, por lo que podemos instrumentar nuestro programa y poner:

- Poner puntos de ruptura (*break points*) con `break`.
- Utilizar el modo de ejecución paso a paso con `step`, `next` y `continue`.
- Visualizar variables con `print` o `disp`.
- Avisos de cambio (*watchpoint*) de variables con `watch`.
- …

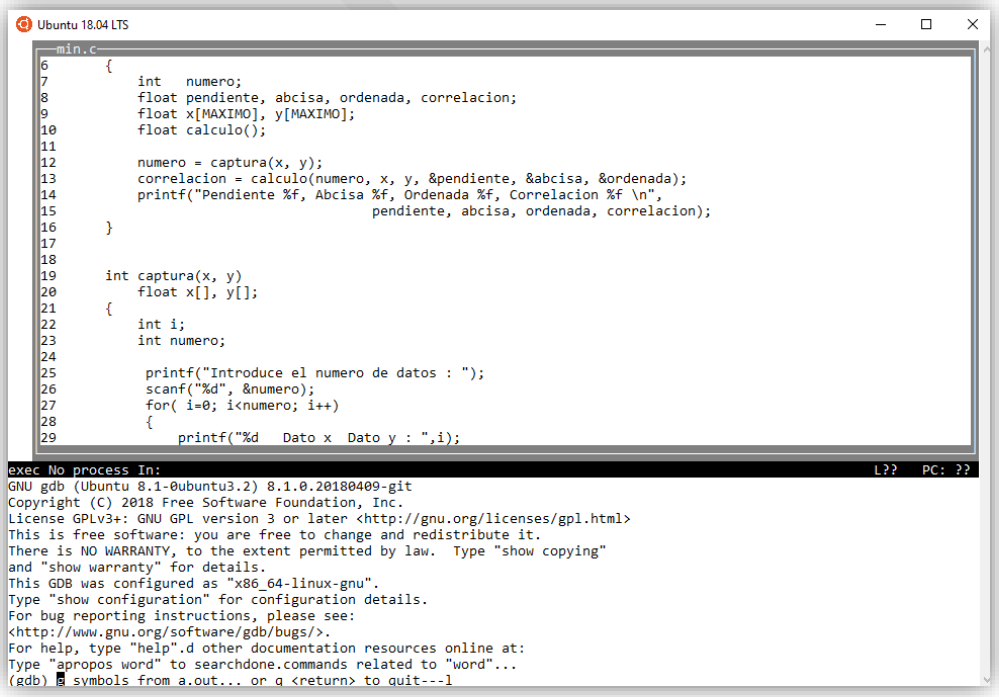
Para especificar el lugar del punto de ruptura debemos usar el comando `break prog.c:6` y la línea de código donde queremos colocarlo, en este caso la 6. También podemos indicar el nombre de una función: `break funcion` o simplemente poner la línea si solo hay un fichero (`b 6`). Una vez puesto el punto de ruptura simplemente ejecutaremos el programa (`run (r)`) y el depurador lo parará llegado ese punto.

Llegados a ese punto puede que nos interese continuar hasta el siguiente punto de ruptura (usaremos `continue (c)`) o simplemente ejecutar la siguiente línea de código (`step (s)`). Si queremos ejecutar una función como si fuera una única línea de código sin entrar en las líneas internas de la misma, usaremos `next (n)`, lo cual es imprescindible si hemos usado un `scanf o printf` en el código.

Cuando el programa esté detenido también podemos ver el valor de las variables, tanto en decimal como en hexadecimal, con el comando `print (p)` o `print/x print/o print/s`, que también sirve para ver punteros, sus indirecciones (`print (*puntero)` o `x puntero`), estructuras (`print estructura.campo`), o arrays (`print cadena[4]`). Si queremos ver continuamente el valor de una variable usaremos `disp variable`.

Si ponemos un perro guardián (`watch variable`) a una variable, cuando esta cambie de valor, el programa se detendrá y pintará los valores viejo y nuevo de esa variable. También se pueden cambiar los valores de las variables usando el comando set: `set (i = 20)`.

Otros comandos útiles son: `backtrace (bt)` o `where`, si una función nos produce un error, podemos ir vuelta atrás; `finish` ejecuta hasta el final de la función actual; `info breakpoints` (abreviado `i b`) muestra información sobre los puntos de ruptura puestos indicados con un código creciente; `delete (d) / clear` borra un *break-point* especificado que puede ser obtenido con el comando anterior.



Si un programa tiene muchos puntos de ruptura, puede ser tedioso ir de uno en uno. Una forma de evitar esto es poner puntos de ruptura condicionales. Por ejemplo:

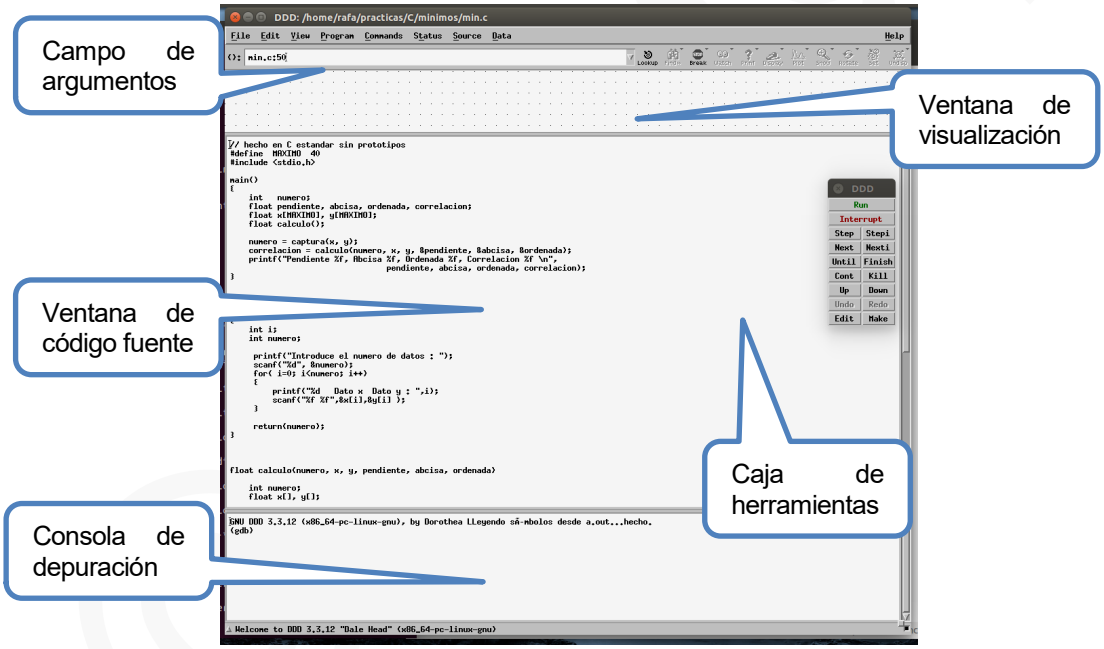
```
break file1.c:6 if i >= TAMANO
```

pondrá un punto en la línea 6 si la variable i es mayor que la dimensión del *array*, útil si hacemos algo parecido a `array[i]` y el tamaño de `array` es `TAMANO`.

Como habrás intuido, es muy útil tener el código fuente a mano, para ello, podemos usar la opción `-tui` a la hora de ejecutar el `gdb`, esta opción dividirá en dos la pantalla y en una se mostrará el código fuente y sus números de línea. Si no, deberemos usar el comando `list (l)` que pondrá en pantalla el código fuente, también vale poner un intervalo de líneas: `3 14` pondrá el código de la línea 3 a la 14. Otra forma de trabajar de manera equivalente es con la aplicación `cgdb` que utiliza la librería de las `curses` (es una forma de trabajar en un terminal de texto cualquiera con algo que parece gráfico pero que se basa en caracteres, la "nueva" versión se llama `ncurses`) con el depurador.

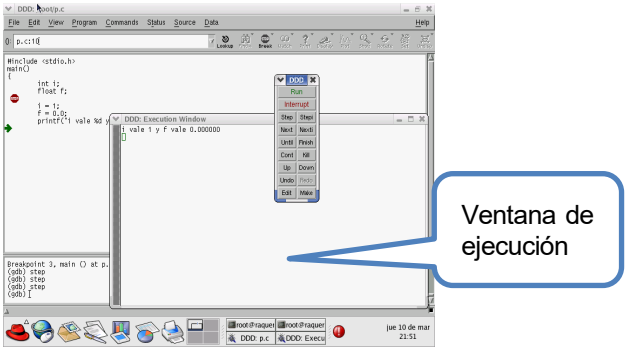
Una tercera forma de trabajar es utilizar el cliente gráfico del programa `ddd`. Para ejecutarlo podemos usar varios modos: `ddd, ddd core, ddd ejecutable`.

Una vez cargado el programa a depurar se podría tener algo como esto:



Arriba un campo para introducir los argumentos al programa, el menú disponible, una ventana de visualización de variables, otra del código, una caja de herramientas y abajo una ventana de introducción de órdenes (por ejemplo, para atender el `scanf`) y depuración.

También se puede arrancar una ventana de ejecución en vez de usar la de órdenes. Se puede forzar desde el menú *view* o con alt-9.



Si utilizas la compilación separada (modular) debes tener presente que todas las funciones deben incorporar la opción

de compilación con depuración, ya que, si no el `gdb/ddd` no podrá mostrar su código fuente ni depurarlas, solo ejecutarlas de un solo paso.

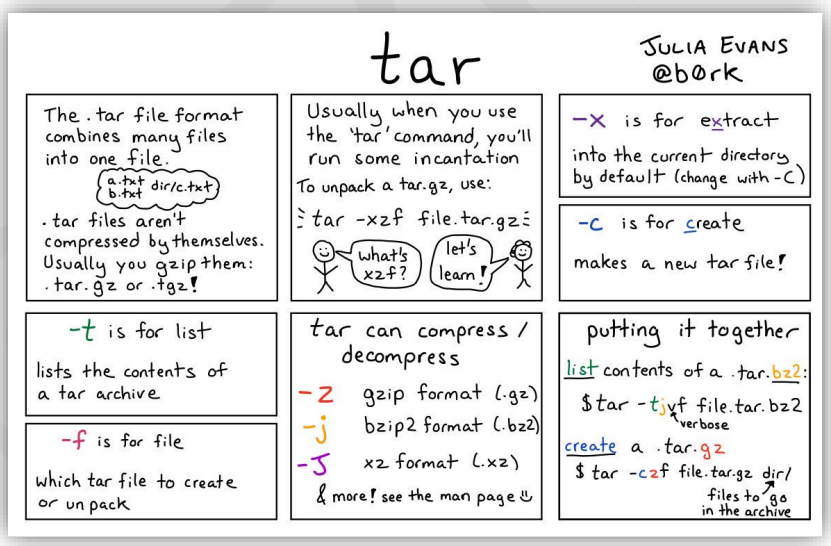
9. Realiza una sesión de depuración usando el programa de mínimos cuadrados convenientemente compilado. Usa poner puntos de ruptura, ejecutar, llegar al punto de ruptura, ver valores de variables (con print y display), tanto de tipo no estructurado (incluyendo punteros) como estructurado y ejecutar paso a paso.

Empaquetado

Por último, y dado que lo vas a usar para mandar las prácticas, existen un par de parejas de comandos para empaquetar/desempaquetar y comprimir y descomprimir ficheros. Para lo primero se utiliza el comando `tar` con diferentes funciones. para lo segundo los comandos `gzip` y `gunzip`.

La arquitectura del primero es la función a realizar, las opciones y los ficheros afectados. Las funciones puedes cualquiera de las letras Acdrtux : concAtenar, crear, diferenciar, añadir, listar, update, extraer (tienen sus variantes largas y más claras). Hay muchas opciones (puedes ver el manual). En el 90% de los casos se utilizan estas versiones de `tar`:

`tar -cvf archivo.tar pepe juan`	crea (función) en el archivo (opción f) los ficheros pepe y juan
`tar --create -f z.tar nano*`	crea z.tar con los archivos que empiezan en nano
`tar -xvf archivo.tar`	extrae los ficheros que hay en archivo.tar
`tar -tvf archive.tar`	lista los ficheros que hay en archivo.tar



El segundo es más sencillo basta con poner el nombre del fichero a comprimir/descomprimir. Por defecto creará un fichero que empieza en z.nombre. Dos ejemplos ilustrativos:

`gzip archivo.tar`	convierte a z.archivo.tar comprimido
`gunzip z.archivo.tar`	convierte de nuevo a archivo.tar descomprimido