# Ayudantía 5 - Estructura de Datos y Algoritmos ELO320
Herramientas

Werner Creixell, Felipe Vera

3 de mayo del 2013

## 1. Compilar un programa de varios archivos .c

Hacer esto es una técnica útil para *modularizar* el código, de modo de tener una funcionalidad separada para distintos archivos .c (Por ejemplo uno puede estar encargado de la parte gráfica, otro algunos algoritmos, y en otro la función *main*).

Esto es común en grandes proyectos con decenas de archivos de código fuente y de salida sólo un ejecutable.

### 1.1. Un .h para cada .c

Los archivos con extensión *.h* generalmente contienen los *defines* y los prototipos de las funciones cuyo cuerpo se encuentra en los archivos *.c*. Estos archivos *.h* deben ser incluidos usando la directiva `#include` en todos los archivos *.c* que empleen dichas funciones.

El comprimido de esta ayudantía incluye un ejemplo. Examine y lea los comentarios contenidos en los archivos `main.c`, `factorial.c` y `factorial.h`.

### 1.2. Compilar

Para compilar cada código fuente *.c* debes usar *gcc* con la opción `-c`, lo cual transforma el código fuente a un *archivo objeto*: una traducción a código máquina de un sólo código fuente. Por ejemplo:

```
gcc -c factorial.c
```

Esto genera un archivo llamado `factorial.o`. Posteriormente, al ejecutar *gcc* con todos los archivos *.o* generados se *linkea* el programa final. Por ejemplo:

```
gcc factorial.o main.o -o programa
```

Si falta incluir un *.o* debería salir un error como el siguiente:

```
main.c:(.text+0x3a): referencia a ‘factorial’ sin definir
```

### 1.3. Linkear librerías externas

En el caso de que se quiera incluir librerías externas (elaborado por terceros al cual normalmente no se tiene acceso al código fuente), se debe agregar opciones al momento de linkear. Por ejemplo para usar las funciones incluidas en `math.h`, se usa la opción `-lm`.

# 2. Debuggeo: Usando DDD

*Debuggear* un programa es importante tanto para comprender dónde un componente (como el valor de una variable o la dirección de un puntero) falla, o se *crashea* el programa. Además se puede interrumpir la ejecución del programa en ciertas partes del programa entre otras opciones. Es una manera mucho más conveniente que llenar el programa de `printfs`.

## 2.1. Compilando el programa para debuggear

Para poder debuggear el programa hay que incluir *información de debuggeo*, el cual consiste en información correspondiente a las líneas de código en ejecución.

Para hacer esto se debe compilar el programa con *gcc* con el comando `-g`. Por ejemplo:

```
gcc -g fuente.c -o programa
```

En el caso de programas de varios códigos fuente, esta opción se coloca al momento de generar los archivos objeto.

## 2.2. DDD

Luego de generar el programa con información de *debuggeo*, se puede ejecutar ddd.

```
ddd programa
```

ddd tiene en la parte superior una barra de herramientas, en medio una muestra del código fuente que se está leyendo, debajo una consola del programa `gdb` (ddd es una interfaz gráfica para gdb), y una barra de herramientas.

### 2.2.1. Insertando breakpoints

Los *breakpoints* son puntos en los cuales se pausa la ejecución del programa. Mientras el programa está pausado se puede:

- Inspeccionar el contenido de cada variable
- Ejecutar el programa instrucción por instrucción, lo cual permite ver el impacto que dicha instrucción genera en las variables del programa.

Para insertar un *breakpoint* se puede:

- Seleccionar una línea y presionar el botón *break* en la barra de herramientas.
- Hacer clic con el botón derecho en una línea y seleccionar *Set Breakpoint*.

Eso coloca un signo "stop" en la línea. Para eliminar el breakpoint, se hace algo similar.

### 2.2.2. Ejecutando el programa

Para ejecutar el programa simplemente se hace clic en el botón **Run** de la barra de herramientas. En el caso de que el programa necesite argumentos, se escribe en la consola de abajo `run argumento1 argumento2 ...`.

### 2.2.3. Observando el valor de las variables

Para observar el valor de una variable, el programa debe estar pausado en un *breakpoint*. En el código fuente se puede hacer clic derecho sobre el nombre de una variable y seleccionar *Watch nombrevariable*. En la parte superior de la ventana aparece un recuadro con el valor actual de la variable.

### 2.3. GDB

Al ocupar un ddd y observar qué es lo que ocurre en la consola inferior, se puede ver el funcionamiento interno de gdb. Este programa se puede ejecutar en modo texto con:

```
gdb programa
```

Para más detalles de los comandos que se pueden ocupar, puedes escribir `man gdb` en la consola.

## 3. Automatizando la compilación: Makefiles

En grandes proyectos que ocupan librerías externas y muchos códigos fuentes, compilar todo manualmente es demasiado difícil. Por eso se inventó el makefile, tanto para compilar normalmente como para ejecutar otras acciones relativas al programa (como instalar al sistema).

Esto es una pincelada sobre *makefiles*. Si quieren más información pueden investigar en internet o en el manual usando `man make`.

Es un archivo llamado `makefile` se escriben las "recetas" necesarias para compilar, y para hacerlo, se ejecuta el programa `make` en la carpeta del *makefile*.

### 3.1. Recetas

Son las instrucciones a seguir para construir una parte del programa o ejecutar una orden.

La primera en ejecutarse al llamar `make` es la receta `all`.

```
(Nombre receta):    (Archivos usados y otras recetas)
                    (Comandos terminal)
```

Típicamente el nombre de la receta es el nombre del archivo resultante luego de "prepararla", como los archivos .o al compilar los .c con el comando -c.

1. **Archivos usados**: Si esos archivos cambian desde la última vez que se compiló el programa, se ejecutan los comandos
2. **Otras recetas**: Etiquetas de otras recetas que se ejecutan recursivamente.
3. **Comandos terminal**: Comandos a ejecutar para "preparar" la receta.

**Nota** Se debe cumplir con las tabulaciones aquí presentadas, de lo contrario `make` no podrá interpretar el *makefile*.

### 3.2. Variables

Se declaran al principio del *makefile*, y puede recurrirse a estas en cualquier parte de la receta: en la de los archivos usados y los comandos de terminal.

```
VARIABLE = main.c

all:    main.o
        gcc -c $(VARIABLE)
```

#### 3.2.1. Comandos shell

También puede almacenarse en esta la salida de un comando en Linux.

```
VARIABLE = $(ls | grep .c)
```

O como comando también.

```
all:    main.o
        gcc -c `ls | grep.c`
```

#### 3.2.2. Macros

Algunas macros útiles son:

- **$@**: Una variable con el nombre de la receta en ejecución.
- **$?**: Archivos que han cambiado desde la última vez que se ejecutó la receta.

Puedes ver el makefile incluido en el archivador para más referencias.

# 4. Profiler: Valgrind

Valgrind es un conjunto de herramientas que sirven como guías para optimizar los programas una vez que el código está escrito y funcional. Valgrind se ejecuta con:

```
valgrind --tool=(herramienta) (programa)
```

También es útil compilar el programa con información de debuggeo antes de ejecutar *Valgrind*.

## 4.1. Memcheck

Memcheck es una herramienta para detectar problemas con variables y memoria. Algunas cosas que Memcheck es posible detectar son:

- Variables no inicializadas (las cuales tienen valores residuales).
- Memoria asignada con `malloc` y no liberada. (contenida en "LEAK SUMMARY")

#### 4.1.1. Otras opciones

Se puede ejecutar con las siguientes opciones:

- `--log-file=report.log` Guarda la salida en un archivo llamado *report.log*
- `--leak-check=full` Detalla dónde ocurren los malloc's que originan las fugas de memoria.

#### 4.1.2. Ejemplo de salida

Un ejemplo de salida es el siguiente:

```
==3481== Conditional jump or move depends on uninitialised value(s)
==3481==    at 0x4E7C4F1: vfprintf (vfprintf.c:1629)
==3481==    by 0x4E858D8: printf (printf.c:35)
==3481==    by 0x40058C: main (memoryleak.c:13)
```

Detalla la línea del código fuente en el cual se originan los errores (si se compila con opción de debuggeo). En este caso el error está en la línea 13 de *memoryleak.c*.

## 4.2. Massif

Es una herramienta para medir cuánta memoria de *heap*, *stack* e instrucciones de procesador se ejecutan en el programa. Se ejecuta de la siguiente manera:

```
valgrind --tool=massif --massif-out-file=report.log
```

Entonces el resultado del *profiling* es guardado en el archivo *report.log*.

### 4.2.1. Interpretando el archivo de salida

Contiene *snapshots* de memoria y cantidad de instrucciones ejecutadas. Para encontrar la máxima cantidad de memoria empleada por el programa, se debe encontrar el *snapshot* en el cual ocurre el máximo `mem_heap`.

### 4.2.2. Otras opciones

Otras opciones de *Massif* son:

- `--stacks=yes` Muestra también la memoria de la zona del *stack* empleada.
- `--time-unit=i` Si se proporciona el argumento *i*, el tiempo de ejecución se mide en ciclos de procesador. Si se proporciona el argumento *ms* se mide en microsegundos.