Depuración

1. Errores de compilación

Son los que hacen que el programa no haya llegado a poder ejecutarse.

Las indicaciones del compilador dicen por dónde anda el error y en qué consiste. Hay que tener en cuenta que la localización exacta del error puede ser anterior a donde dice el mensaje y tampoco su explicación es totalmente fiable. Además un error puede hacer que el compilador se líe completamente, marcando después otros errores que no lo son, por lo que puede ser conveniente volver a intentarlo después de corregir el primer error, salvo que los otros se vean claros.

En caso de duda respecto a la localización del error puede ser útil el desarrollo incremental, comentado en el apartado 5.

2. Errores de ejecución

En caso de que no funcione bien hay que asegurarse de comprobar todos los avisos (*warning* en inglés) dados en la fase de chequeo o compilación del programa, aunque no estén marcados como graves.

Si tienes más de un compilador puedes probar, porque puede que den mensajes distintos.

Para comprobar cómo son los fallos es conveniente hacer varias pruebas y ver qué sale: meter varios conjuntos de datos, meter casos límite, mirar el programa y hacer pruebas que seguro que pasen por todos los casos previstos. Esto puede dar una idea de por dónde anda el error.

Las técnicas que vamos a comentar para encontrar los fallos son:

- Revisión de código
- Uso de depurador
- Programación incremental
- Mensajes de error
- Mensajes de control

Si todo falla, siempre se puede probar a hacer el programa de otra manera. Conviene recordar que los programas no tienen solución única.

También, si no estás en el último minuto, relájate un poco y piensa en otra cosa. En un segundo vistazo habrá mejores posibilidades.

Con práctica se va cogiendo agilidad para encontrar errores.

No esperes al final de curso para empezar a practicar.

3. Revisión de código

3.1. Revisión de estilo

Debería haberse hecho antes de la compilación, pero si el error no se ha detectado, hay que volver a hacerla.

Cada vez que encuentres un fallo, conviene que te vayas apuntando el tipo de fallo que es, porque las personas tendemos a equivocarnos en lo mismo. Esa lista de tus fallos típicos la puedes usar para localizar errores.

3.2. Revisión de errores frecuentes

Además de los que tengas previstos en tu lista de fallos típicos, comentada en el apartado anterior, estos son algunos errores frecuentes:

- Código al que no se llega porque hay alguna instrucción que se lo salta (condición de ciclo o de condicional que no se cumple nunca)
- Usar una variable equivocada
- Confundir & con && ó `|` con `||`
- Confundir = con ==
- Poner ; justo detrás de `for` (**sí** hay que ponerlo dentro) o detrás de `while`, o de la definición de una función. El compilador no avisa, así que hay que comprobarlo manualmente.
- Poner ; en instrucciones que empiezan por #
- Usar = en la instrucción `#define`
- No poner ; en instrucciones normales (asignación, llamada a función)
- Problemas con la precedencia entre operadores. Nunca se producirá un error de compilación. El consejo es usar siempre paréntesis, a no ser que estemos **absolutamente** seguros.
- Utilizar formatos equivocados en `printf` o `scanf`.
- No poner & en un `scanf`
- Problemas con las llaves (falta o sobra alguna).
- No poner `break` al final de cada caso de un `switch`.
- Confundir el orden o tipo de argumentos de una función entre la llamada y la definición.

Si se cuelga el programa, puede ser debido a un ciclo que no sale nunca (revísalos, puedes usar alguna de las técnicas comentadas más adelante), o porque está esperando una entrada de teclado (instrucción `scanf` que no va precedida del correspondiente `printf`).

3.3. Verificación

Cada estructura de un programa debe estar claro lo que hace, o porque es obvio viéndola o porque está indicado con comentarios. Las estructuras posibles son: ciclos, condicionales y secuencia de instrucciones simples. Se deben verificar las estructuras para ver si hacen lo que se supone. En el caso de condicionales, en todos los casos. En el caso de ciclos hay que ver además si termina en todos los casos.

3.4. Comprobación de cumplimiento

Una revisión útil es coger cada bloque de instrucciones o función entera y definir lo que hace, **no dándolo por supuesto**, sino explicándolo de nuevas a partir de lo que está escrito en el programa, y comprobar si ensamblando todas las cosas que hace lo que está allí puesto, se llega a lo que de verdad tenía que hacer.

3.5. Ejecución manual

Otra revisión es ir ejecutando nosotros el programa *a mano*, para ver si sale bien. Aprovechar para plagarlo de comentarios explicativos, porque al hacer alguna explicación, puede que se vea claramente cuál es el error. Los resultados pueden usarse para hacer una "tabla de ejecución", que se puede usar en otros apartados.

En una tabla de ejecución cada fila se corresponde con una línea ejecutable del programa y cada columna con una variable. Para las filas pueden usarse todas las instrucciones cada vez que se pase por ellas o sólo algunas. Para las columnas pueden usarse todas las variables o sólo las más significativas. A continuación se va ejecutando el programa *a mano* y calculando nosotros el valor de cada variable.

4. Uso de depurador

El manejo del depurador se verá en prácticas. Lo fundamental es ejecutar línea a línea y ver valores de variables, pero puede que te venga bien:

- usar puntos de parada (*breakpoint* en inglés)
- cambiar valores de variables
- ver varios valores simultáneamente
- ver el resultado de un cálculo con las variables del programa

Pregunta al profesor en cualquier sesión todas las dudas que te queden al respecto.

Con el depurador se puede ir ejecutando instrucción a instrucción y verificando los valores de las variables. Puedes verificar las variables calculando sus valores correctos sobre la marcha o preparando una tabla de ejecución (apartado 3.5).

En principio conviene ir línea a línea y pasando por todas las funciones programadas. Si alguna función ya ha sido comprobada y es de confianza, puede saltarse, es decir ejecutarla entera de una vez. Igualmente, si algún bloque de instrucciones está ya comprobado, puede ejecutarse sin parar, usando puntos de parada. Un punto de parada se coloca donde queramos y a continuación se manda ejecutar el programa seguido (no línea a línea); parará al llegar a la línea indicada.

En caso de duda, se puede cambiar directamente el valor de una variable para ver qué pasaría a partir de ese punto con el nuevo valor.

5. Programación incremental

Es aconsejable empezar con ella desde el principio, pero se puede usar también a posteriori.

Cuando se usa desde el principio, esto consiste en empezar por el caso más simple e ir añadiéndole cosas hasta llegar a lo pedido.

Cuando se usa para buscar errores, consiste en quitar trozos de programa (mediante comentarios o creando un fichero nuevo para estas pruebas) para ver si lo que queda funciona. Puede ser también una

simplificación (por ejemplo unas pocas variables en vez de un tensor). Añadiendo cada vez una parte de programa se puede llegar a descubrir donde está el problema.

Una técnica que puede ser conveniente es probar cada función por separado, haciendo un programa que sólo llame a esa función y ver qué pasa. También se puede hacer con bloques de instrucciones más pequeños .

6. Mensajes de error

Se pondrá al principio del programa la instrucción `#include<assert.h>`. A lo largo del programa se pondrán comprobaciones de condiciones que tienen que darse usando la instrucción

`assert(`*`condición que tiene que ser`*`)`

Por ejemplo:

- que un número no pueda ser negativo (`assert(numero>=0)`)se pondría justo antes de hacerle una raíz cuadrada
- que un índice no excede del límite posible para un tensor[1] se pondría justo antes de usarlo para indicar un elemento del tensor

Esto se puede vigilar bien usando la tabla de ejecución comentada en el apartado 3.5. Luego al ejecutar el programa, si una comprobación falla, parará con un mensaje indicando qué condición ha sido la que ha fallado. Si con eso todavía no acotas la posición del error, pon más mensajes.

7. Mensajes de control

Pon al principio de tu programa lo siguiente:

```
#define DEPURA
```

Luego, a lo largo del programa, coloca instrucciones que escriban los valores de las variables que estés manejando, de la siguiente forma:

```
#if defined(DEPURA)
printf(....);
fflush(stdout)
#endif
```

De esta forma al ejecutar irán saliendo los valores de las variables. Comprueba que son los que tú esperas, lo cual puedes hacer usando una tabla de ejecución, comentada en el apartado 3.5.

También pueden usarse mensajes meramente informativos del tipo de "empezando el ciclo de ...", "terminado el ciclo ...", "entrando en la función ...", etc.

Si en algún caso no sale lo esperado, es que antes de ese mensaje se ha producido un error. Si la localización no está clara, usa más mensajes.

Cuando ya no quieras los mensajes de depuración simplemente quita la línea inicial `#define DEPURA`. El resto se cancelará automáticamente. Si sale demasiada información puedes quitar algunos mensajes. Puedes, o borrarlos, o inutilizarlos metiéndolos en comentarios, o usar `#define DEPURA1, #define DEPURA2`, etc. y usar DEPURA1 o DEPURA2 o lo que sea en cada printf, de forma que quitando el `#define` inicial, anulas todo el grupo.

[1] Usaremos tensor como sinónimo de array