Arquitectura de sistemas

Abelardo Pardo

University of Sydney
School of Electrical and Information Engineering
NSW, 2006, Australia
Autor principal del curso de 2009 a 2012

Iria Estévez Ayres

Damaris Fuentes Lorenzo

Pablo Basanta Val

Pedro J. Muñoz Merino

Hugo A. Parada

Derick Leony

Universidad Carlos III de Madrid
Departamento de Ingeniería Telemática
Avenida Universidad 30, E28911 Leganés (Madrid), España

Capítulo 12. Las opciones más utilizadas del compilador gcc

Tabla de contenidos

12.1. De un fichero de código a un programa
12.2. Mostrando todas las advertencias
12.3. La opción de depuración
12.4. Optimización de código
12.5. Definiendo símbolos
12.6. Código en varios ficheros
12.7. Compilando en dos pasos
12.8. Paso de argumentos a un programa
12.9. Corrección de errores de compilación
12.10. Bibliografía de apoyo
12.11. El Preprocesador
- 12.11.1. La directiva `#include`
- 12.11.2. La directiva `#define`
- 12.11.3. Las directivas `#ifdef`, `#else` y `#endif`
- 12.11.4. Definición de "macros" con la directiva `#define`

12.12. Los errores y advertencias del compilador
- 12.12.1. Errores y advertencias
- 12.12.2. Mensajes más comunes del compilador
- 12.12.3. Error en la ejecución

El compilador de C de GNU **gcc** es la aplicación que, dado un conjunto de ficheros de código C, genera un programa ejecutable. Como el resto de aplicaciones, se puede invocar desde el intérprete de comandos. Con la opción **--version** el compilador tan sólo muestra información sobre su versión. Prueba el siguiente comando en el terminal:

```
$ gcc --version
```

Como todo comando, **gcc** también tiene su página de manual en donde se explica su funcionamiento así como las opciones con las que se puede modificar su comportamiento. Para ver la complejidad de esta herramienta mira esta página de manual con el comando:

```
$ man gcc
```

12.1. De un fichero de código a un programa

Crea una carpeta en tu cuenta de usuario con nombre `Lab2` y descarga en ella el fichero `main_es.c`. Abre el fichero con el editor y comprueba que contiene la función `main` por la que comienza la ejecución del programa así como una función auxiliar. A continuación ejecuta el comando para transformar este fichero de código en un programa:

```
$ gcc main_es.c
```

Comprueba que en el directorio actual se ha creado el fichero `a.out`. El compilador está diseñado para que si no se especifica el nombre del programa a crear, por defecto lo deja en `a.out`. Este

fichero se puede ejecutar como cualquier otro comando, con la salvedad de que por estar ubicado en una carpeta del usuario y no en las que el sistema busca los programas, hay que ponerle el prefijo "./", esto es:

```
$ ./a.out
```

La primera opción que vamos a utilizar es precisamente para poder escoger el nombre del ejecutable que produce el compilador. Esta opción es **-o** seguida (separado por un espacio) del nombre del fichero que queremos crear. Por ejemplo, si queremos llamar al programa `main_es`, el comando a ejecutar es:

```
$ gcc -o main_es main_es.c
```

Tras ejecutar este comando comprueba que tienes ***dos*** ejecutables en el directorio actual. Borra el que ya no necesitas.

12.2. Mostrando todas las advertencias

Al contrario que otros compiladores como el de Java, el de C no realiza una comprobación exhaustiva de que el programa cumple estrictamente con la definición del lenguaje. Cuando en un programa en C se omiten ciertos detalles, el compilador asume un comportamiento por defecto y genera igualmente un ejecutable. Pero esta política es peligrosa si el programador no conoce en detalle estos comportamientos que asume el compilador. Por este motivo se ofrece la opción **-W** que se puede incluir especificando qué tipo de comprobaciones queremos que haga el compilador.

Borra el ejecutable de tu directorio. Compila de nuevo el programa anterior pero esta vez solicitando que se muestren todas las advertencias con el siguiente comando:

```
$ gcc -Wall -o main_es main_es.c
```

Verás que, a pesar de no haber cambiado el código, el compilador ahora nos muestra ciertas advertencias pero el ejecutable se ha generado igualmente. En este caso el programa se comporta exactamente como esperamos, pero toda advertencia que muestra el compilador es sobre algún aspecto del código que no está perfectamente especificado y que por tanto puede tener consecuencias en tiempo de ejecución. Lee detenidamente el mensaje de las advertencias y modifica el código para que desaparezcan. No hagas cambios aleatorios en el código, pues así no sabrás cual es la razón exacta para la advertencia.

Te recomendamos que ***todas las compilaciones*** que hagas durante el curso las hagas con la opción **-Wall**. Nosotros lo haremos a la hora de evaluar la calidad de tu código.

12.3. La opción de depuración

El ejecutable que genera el compilador contiene información sobre los símbolos que utiliza. El comando **nm** muestra estos símbolos. Lee su página de manual y utilízalo para mostrar los símbolos por pantalla. ¿Qué diferencia hay en los datos mostrados cuando se utiliza la opción **-a**?

Detectar errores de ejecución en un programa en C que ha compilado sin problemas es la parte más costosa. Para ello se utiliza el ***depurador***, un programa capaz de ejecutar un programa línea a línea mostrando al mismo tiempo el código fuente y el valor de las variables. Pero para hacer esto, el depurador necesita que el ejecutable incluya qué ficheros son los que tienen el código fuente. Esta información no es necesaria para la ejecución normal del programa, y por tanto, para incluirla en el ejecutable, se utiliza la opción **-g**. Compila de nuevo el programa esta vez con la opción **-g**:

```
$ gcc -g -Wall -o main_es main_es.c
```

Utilizando diferentes valores de la opción **-o** genera dos ejecutables, uno con la información para la depuración y otro sin ella. Comprueba con el comando **ls** que tienen diferentes tamaños. A continuación utiliza el comando **nm** con las opciones **-al** para mostrar los símbolos. ¿Qué diferencia hay entre ambos ejecutables?

Puedes capturar la salida del comando **nm** en sendos ficheros y compararlos, o usando el editor (lo puedes abrir en dos ventanas separadas) o con el comando **diff** (lee su página de manual). Busca la aparición del nombre del fichero que contiene el código fuente.

12.4. Optimización de código

El intérprete de comandos BASH permite medir el tiempo que tarda en ejecutarse un comando simplemente anteponiendo el comando **time** al comienzo de la linea. El tiempo se divide en tres categorías: tiempo total, del usuario y del sistema. Las medidas son aproximadas, con lo que la suma de las dos últimas no tiene por qué dar como resultado la primera. Ejecuta el programa con el comando

```
$ time ./main_es
```

y anota el resultado.

Una de las funcionalidades más potentes del compilador es la de "optimizar el código". Existen múltiples técnicas para analizar y transformar el código generado inicialmente por el compilador de forma que se obtenga un mejor rendimiento o en tiempo de ejecución, o en uso de memoria. Para que el compilador aplique estas técnicas se debe incluir la opción **-O**. Compila de nuevo el programa con esta opción y ejecútalo de nuevo con el comando **time** como prefijo. Compara los dos tiempos de ejecución. ¿Te parece efectivo el compilador optimizando el código?

12.5. Definiendo símbolos

La función `print_line` imprime una línea con el símbolo "-" repetido tantas veces como indica su único parámetro. Ese valor se le pasa a través de la variable `line_width` declarada y asignada en la primera línea de la función `main`. Si el programa tuviese que ejecutarse en dispositivos con ancho de pantalla de 20, 40 y 60 caracteres, ¿cómo obtendrías estas tres versiones? Es decir, hay que obtener tres ejecutables del mismo programa, pero que impriman líneas de diferente longitud.

El compilador dispone de una funcionalidad que es muy útil en estos casos. Edita el programa, borra la primera línea de la función `main` donde se define y asigna el valor a `line_width`. Modifica las dos líneas que llaman a la función `print_line` tal que sea:

```
print_line(LINEWIDTH);
```

El programa es aparentemente incorrecto, pues hay un símbolo, concretamente `LINEWIDTH` que se utiliza pero que no se declara en ningún sitio. Compila ahora el programa pero utilizando la opción **-D** seguida de la definición **LINEWIDTH=80** tal y como se muestra en el siguiente comando:

```
$ gcc -DLINEWIDTH=80 -o main_es main_es.c
```

Como puedes comprobar, el compilador no informa de ningún error y el ejecutable se crea normalmente. La opción **-D** seguida del nombre de un símbolo y de (opcionalmente) un signo igual y un valor hace que el compilador pre-procese el código fuente y toda aparición del símbolo la reemplace por el valor dado. Ojo, que lo que hace el compilador es un ***reemplazo textual***, en otras palabras, el programa no tiene ningún símbolo con nombre `LINEWIDTH` sino que en el programa hay dos líneas que en la compilación anterior se han procesado como

```
print_line(80)
```

Prueba a re-compilar con diferentes valores del símbolo `LINEWIDTH` y comprueba que se imprimen diferentes longitudes de línea. ¿Qué sucede si omites en el comando de compilación el signo igual y el valor? Para distinguir en el código los símbolos del programa de aquellos que se procesan con este reemplazo textual, se utiliza la convención (que no obligación) de escribirlos todos en letras mayúsculas.

12.6. Código en varios ficheros

El compilador puede generar un ejecutable con código que está en múltiples ficheros. Una condición necesaria es que haya una única definición de la función `main`. Toma el fichero de código del programa y pon cada función en un fichero diferente (por ejemplo: `main.c`, `fichero1.c`, `fichero2.c`). Compila los tres ficheros a la vez con la opción **-Wall** y modifica el código para que compile sin advertencias. ¿En qué ficheros necesitas replicar la línea `#include <stdio.h>`?

12.7. Compilando en dos pasos

Hasta ahora el proceso de compilación que hemos tratado se hace en un solo paso. Sin embargo, este se divide en dos pasos: uno de compilación y otro de enlazado. El paso de compilación crea codigo ensamblador y lo convierte a código objeto. Para realizar este paso debes indicar al compilador que detenga el proceso antes de enlazar. Esto se hace adicionado la opción `-c` al comando `gcc`, así:

```
$ gcc -c filename.c
```

Usualmente un programa C se compone de varios ficheros `.c`. Para compilarlo, bastará con compilar cada fichero `.c` por separado, obteniendo como resultado un fichero `.o` para cada uno. Toma cada fichero de código `.c` generado en la sección anterior, aplica el primer paso del proceso de compilación, y verifica que hayas obtenido un fichero `.o` por cada fichero `.c` compilado.

Ahora ya puedes generar el programa ejecutable, para ello bastará con ejecutar el paso de enlazado. Este paso enlaza tu programa con otros programas que contienen funciones que tu programa usa. Para ello debes adicionar al comando `gcc` la opción `-o` con el nombre del programa ejecutable, seguido de la lista de ficheros `.o` obtenidos en el paso de compilación.

```
$ gcc -o program filename1.o filename2.o filename3.o
```

Ejecuta el programa obtenido y verifica los resultados. Compilar los programas por separado es útil por ejemplo, cuando se han realizado cambios en un solo fichero `.c`, solo sería necesario compilar dicho programa y enlazarlo con los demás `.o`. Prueba a realizar un cambio en uno de tus ficheros `.c` compila este fichero con la opción `-c`, enlaza el fichero `.o` con los otros ficheros objeto (compilando con la opción `-o`), y ejecuta el programa. Consiseras útil esta opción? La usarías en progrmas más complejos?

12.8. Paso de argumentos a un programa

El programa creado por el compilador se puede ejecutar como un comando más por el intérprete, pero ¿se pueden pasar argumentos? Prueba a ejecutar de nuevo el programa pero añadiendo una lista de palabras separadas por espacios. ¿Qué imprime el programa? Analiza el primer bucle `for` y deduce qué se almacena en las variables `argc` y `argv`.

12.9. Corrección de errores de compilación

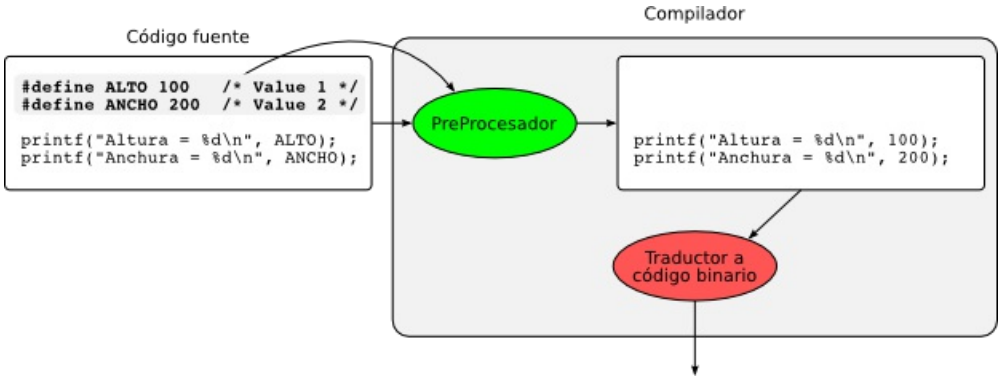
Descarga el programa bogus.c que debe imprimir por pantalla el cuadrado de los números del 1 al 10. Debes modificar el programa para que compile sin advertencias con la opción **-Wall**. Comprueba después que ejecuta correctamente.

12.10. Bibliografía de apoyo

- ***How the compiler works***: " An Introduction to GCC"by Brian J. Gough, foreword by Richard M. Stallman, pp. 89-93

12.11. El Preprocesador

El preprocesador es un programa que forma parte del compilador y que "prepara" o modifica el código fuente antes de ser traducido a código binario. Los cambios los hace interpretando aquellas líneas del código que comienzan por el símbolo "#". El uso de estas directivas son tan comunes en los programas en C que parece que forman parte del lenguaje, pero en realidad son parte de un lenguaje que sólo entiende el procesador. La siguiente figura ilustra como se procesa un fichero de código fuente en realidad por el compilador.

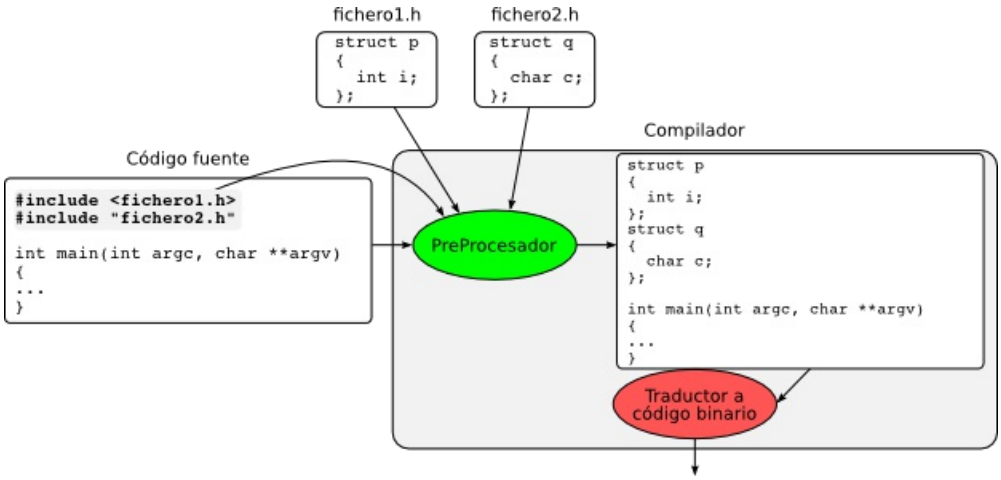


Como se puede comprobar, las dos líneas que comienzan por "#" han desaparecido, al igual que los comentarios. Al traductor a código binario le llega fichero sin ninguna directiva ni comentario.

El preprocesador puede ser utilizado de forma independiente del compilador mediante el comando `cpp`. Abre un terminal de comandos y consulta la página de manual de este comando. Como puedes comprobar, es un programa que puede procesar un amplio catálogo de directivas. De todas ellas, en este documento se describen las más utilizadas.

12.11.1. La directiva `#include`

La directiva `#include` debe ir seguida del nombre de un fichero y su efecto es de reemplazar esta línea en el código fuente por el contenido del fichero que se especifica. La siguiente figura ilustra un ejemplo de esta directiva.



Con esta directiva debes tener en cuenta lo siguiente:

- Si el fichero que va a continuación de `#include` está rodeado por "<" y ">", el preprocesador lo busca en los directorios internos del sistema. Esta versión se utiliza por tanto para incluir ficheros dados por el sistema.

- Si el fichero está rodeado de doble comilla, entonces se busca en el directorio en el que se está compilando. Esta versión de la directiva se utiliza para incluir ficheros que ha escrito el usuario. La opción **-L** del compilador permite especificar una lista adicional de directorios en los que buscar este tipo de ficheros.
- La extensión "`.h`" se suele utilizar para ficheros que se incluyen en un programa con esta directiva
- Antes de incluir el contenido del fichero, se procesan las directivas contenidas en su interior. Esto permite que un fichero que se incluye en un programa incluya a su vez otros.
- Este tipo de ficheros suele incluir definiciones que se necesitan en más de un fichero. En lugar de repetir estas definiciones en los ficheros de código (con extensión "`.c`"), se pasan a un fichero con extensión "`.h`" que se incluye en todos ellos.
- Los ficheros incluidos con la directiva `#include` no se escriben en el comando que invoca al compilador. Su utilización la decide el preprocesador cuando encuentra la directiva `#include`.

12.11.2. La directiva #define

La directiva `#define` tiene dos versiones. Si va seguida de una única cadena de texto como por ejemplo

```
#define SISTEMA_MAEMO
```

el preprocesador simplemente anota internamente que este símbolo está "definido". En la sección 12.11.3 veremos otra directiva para consultar qué símbolos están definidos.

La segunda versión de esta directiva es cuando va seguida de dos cadenas de texto. En este caso, a partir de ese punto, el preprocesador reemplaza toda aparición de la primera cadena por la segunda. El siguiente ejemplo define el símbolo `CARGA_MAXIMA` para que se reemplace por el valor `1500`.

```
#define CARGA_MAXIMA 1500
```

El efecto de esta directiva es ***idéntico*** al uso de la opción **-D** del compilador. En realidad, cuando se invoca el compilador con la opción **-Dnombre=valor**, esta definición es equivalente a que el procesador encontrase la línea `#define nombre valor`.

Con la directiva `#define` debes tener en cuenta lo siguiente:

- Esta directiva suele estar al comienzo de los ficheros de código, o si se necesitan en varios ficheros, en un fichero de definiciones con extensión "`.h`" que se incluye en otros ficheros.
- Para diferenciar en el código los símbolos normales de un programa de aquellos que han sido definidos por la directiva `#define` y que van a ser reemplazados por sus equivalentes por el preprocesador, estos últimos se suelen escribir siempre con mayúsculas (esto es una convención, el preprocesador no realiza ningún tipo de comprobación).
- El reemplazo del símbolo por su valor se realiza en todo el texto de un fichero. Esto incluye también las propias directivas del preprocesador. En el siguiente ejemplo

```
#define UNO 1
#define OTRO_UNO UNO

int main(argc, char *argv[])
{
    printf("%d\n", OTRO_UNO);
```

```
}
```

el programa imprime el número uno. Es decir, la segunda directiva `#define` se procesa como `#define OTRO_UNO 1` al reemplazarse `UNO` por la definición de la línea anterior.

12.11.3. Las directivas #ifdef, #else y #endif

En la sección 12.11.2 hemos visto como el preprocesador mantiene un conjunto símbolos definidos, y algunos de ellos deben ser sustituidos por sus valores equivalentes. El preprocesador también ofrece un mecanismo por el que una porción de código de un programa se puede ocultar o considerar dependiendo del valor de alguno de los símbolos definidos con la directiva `#define`. La estructura de esta construcción es la siguiente:

```
#ifdef SIMBOLO
  /* Bloque de código 1 */
  ...
#else
  /* Bloque de código 2 */
#endif
```

Cuando el preprocesador encuentra la primera directiva `#ifdef SIMBOLO`, si `SIMBOLO` está definido, pasa el bloque de código 1 al compilador (hasta la directiva `#else`) y elimina el bloque de código 2 (entre las directivas `#else` y `#endif`. De forma análoga, si `SIMBOLO` no está definido, el bloque de código 1 se elimina y el compilador sólo recibe el bloque de código 2. Esta construcción es similar al `if/then/else` en C, pero la diferencia es que esta la interpreta el preprocesador cuando se compila. La diferencia está en que si el bloque de código que se ignora contiene un error de sintaxis, el compilador generará el programa igualmente, pues no llega a procesar ese código.

Esta directiva se utiliza cuando se quiere mantener dos versiones de un programa que se diferencian únicamente en un reducido número de líneas de código. Las dos versiones pueden coexistir en el código fuente pero rodeadas de esta directiva. Al compilar se utiliza entonces la opción `–Dnombre=valor` para seleccionar los bloques de código pertinentes y generar el ejecutable.

El siguiente ejemplo muestra el uso de esta directiva para escribir dos versiones de un mensaje de bienvenida a un sistema con dos posibles versiones. Si el símbolo `MAEMO` está definido (por ejemplo al compilar **gcc -DMAEMO ...**) al ejecutar se imprime un mensaje, y si no está definido este símbolo, se imprime un mensaje alternativo.

```
#ifdef MAEMO
    printf("Bienvenido al sistema Maemo\n"); /* Código para la versión MAEMO */
#else
    printf("Bienvenido al otro sistema\n"); /* Código para la otra versión */
#endif
```

12.11.4. Definición de "macros" con la directiva #define

La directiva `#define` tiene una funcionalidad extra que puede utilizarse para definir lo que se conoce como "macros". El reemplazo que hace el preprocesador del símbolo por su equivalente puede incluir parámetros. En el siguiente ejemplo se define una macro para reemplazar el símbolo `DEMASIADO_GRANDE(v)` la comparación de `v` dada con el valor 1000.

```
#define DEMASIADO_GRANDE(v) (v >= 1000)
```

La macro `DEMASIADO_GRANDE(v)` se puede utilizar en el código con un nombre de variable en lugar de `v` que será utilizado al reemplazarse el símbolo por su equivalente tal y como se muestra en el

siguiente ejemplo:

```
int i;
if (DEMASIADO_GRANDE(i)) /* Código fuente */
{
  ...
}

if ((i >= 1000))  /* Código recibido por el traductor */
{
  ...
}
```

12.12. Los errores y advertencias del compilador

El compilador es el programa que, dado un conjunto de ficheros de código, los traduce y genera un fichero listo para ejecutar. Pero para que esta traducción sea posible, el programa debe cumplir con los requisitos que impone el lenguaje de programación. El tipo de comprobaciones que se hacen en esta traducción depende tanto del lenguaje de programación como del propio compilador. Cuando se detecta en el programa algo que no está permitido por las normas del lenguaje, el compilador muestra un mensaje y se detiene la traducción.

Estos mensajes no explican en detalle la causa por la que aparecen, sino la anomalía puntual que ha detectado el compilador. Por este motivo, puede ser difícil identificar la verdadera razón detrás de un error mostrado por el compilador.

12.12.1. Errores y advertencias

El lenguaje de programación C está definido de tal forma que hay numerosas decisiones que el compilador adopta de forma automática ante la ausencia de información. En otras palabras, el compilador toma decisiones sobre cómo traducir el código que el programador no ha escrito en el código. Esto, en ciertas situaciones, es cómodo, pues hace que los programas puedan ser escritos más rápido, pero sólo programadores con experiencia suelen aprovechar esta característica. Lo más recomendable es utilizar una opción en el compilador para que notifique aquellos casos en los que está tomando decisiones implícitas. En el compilador **gcc** esta opción es **-Wall**.

El compilador muestra dos tipos de anomalías: errores y advertencias. Un error es una condición que impide la obtención del programa final. Hasta que no se solventen todos los errores, el compilador no puede terminar su trabajo. Los primeros errores que se muestran son los más fiables porque se intenta traducir todo el programa hasta el final, con lo que es posible que haya errores que se deriven de otro anterior. Por tanto, si se arreglan los primeros errores, es recomendable compilar de nuevo para ver si como consecuencia han desaparecido también otros errores posteriores.

Las advertencias son mensajes que muestra el compilador sobre situaciones "especiales" en las que se ha detectado una anomalía pero que, asumiendo ciertas condiciones, la traducción del programa continúa. Por tanto, la compilación de un programa no termina mientras tenga errores pero, sin embargo, se puede obtener un programa traducido aunque el compilador muestre advertencias. Cuando se comienza a programar en C y hasta que no se tenga un nivel de experiencia elevado en la codificación de aplicaciones complejas (numerosos ficheros de código) la recomendación es:

> Compilar ***siempre*** con la opción **-Wall** y no considerar el programa correcto hasta que no se hayan eliminado ***todas*** las advertencias.

12.12.2. Mensajes más comunes del compilador

A continuación se describen los mensajes más comunes que muestra el compilador. Se ofrecen

ejemplos de código para ilustrar las causas del error, pero debe tenerse en cuenta que cada programa tiene una estructura diferente, y por tanto, el error puede aparecer por causas de alto nivel diferentes.

1. `variable' undeclared (first use in this function)

   Este es uno de los más comunes y a la vez más fáciles de detectar. El símbolo que se muestra al comienzo del mensaje se utiliza pero no ha sido declarado. Sólo se muestra la primera de las apariciones.

2. warning: implicit declaration of function `...'

   Esta advertencia aparece cuando el compilador encuentra una función que se utiliza en el código y de la que no tienen ningún tipo de información hasta el momento. El compilador asume que la función devuelve un entero, y continúa.

   Estas advertencias son las que se recomienda eliminar antes de probar la ejecución del programa. Dependiendo de la situación, hay varias formas de eliminar este mensaje. El compilador necesita saber cómo se define la función que ha encontrado. Para ello, en un lugar anterior en el fichero se debe incluir, o la definición de la función o su prototipo (el tipo de dato que devuelve, seguido de su nombre, los paréntesis con la definición de los parámetros y un punto y coma).

   Si esta información (la definición de la función) se necesita en múltiples ficheros, entonces, en lugar de escribir la misma información en cada uno de ellos se recomienda incluir el prototipo en un fichero que es luego incluido en donde se necesite a través de la orden `#include` del pre-procesador. Recuerda que el código de una función sólo puede estar presente en un único fichero de código. Si el compilador se lo encuentra en más de un fichero, entonces se producirá un error por haber definido una función múltiples veces.

   También puede darse el caso de que la información que se solicita esté contenida en un fichero a incluir por tratarse de una función en una biblioteca. Para consultar si la función está en una de las bibliotecas estándar de C se recomienda utilizar el comando **man** seguido por el nombre de la función. Si es parte de una biblioteca, en la misma página de manual se muestra qué fichero hay que incluir en el código.

3. warning: initialization makes integer from pointer without a cast

   Esta advertencia nos dice que hemos utilizado un puntero en un contexto en el que se espera un entero. Esto está permitido en C y el compilador sabe qué hacer para traducir el código, pero igualmente se notifica por si se quiere "confirmar" esta operación mediante un *cast*. El siguiente código muestra un ejemplo que produce esta advertencia.

```
int main (void)
{
  char c = "\n";  /* ¿incorrecto? */
  return 0;
}
```

   En la primera línea se asigna un puntero a letra (pues una cadena entrecomillada se almacena por debajo como una dirección de memoria, que es la que contiene la primera letra) a una letra. A pesar de que `c` está definida como una letra, el compilador la trata como un entero, lo cual es legal. La cadena de texto `"\n"` se toma como un puntero, y de ahí se deriva el mensaje de advertencia. Se está asignando un puntero a una variable que no lo es y que puede ser considerada como un entero, y además, no se está utilizando un *cast*.

   En la mayoría de los casos esta asignación es en realidad un error del programador y por tanto necesita ser corregido. Pero es posible que se quiera manipular un puntero como un número tal y

necesita ser corregido. Pero es posible que se quiera manipular un puntero como un número tal y como se muestra en el siguiente programa:

```
int main (void)
{
  int value = "\n"; /* Asignación correcta */
  return 0;
}
```

Supongamos que en este caso el programador sí necesita almacenar ese puntero en una variable entera. El compilador igualmente imprime la advertencia. Para confirmar al compilador que realmente este es la asignación que se quiere hacer se puede utilizar un *casting*, esto es, poner entre paréntesis como prefijo de un símbolo el tipo de datos al que queremos transformarlo. Una vez añadido este prefijo, el compilador entiende que esta asignación es realmente lo que se quiere hacer y no se muestra la advertencia.

```
int main (void)
{
  int value = (int)"\n"; /* Asignación correcta y confirmada por el programador
                         mediante el casting */
  return 0;
}
```

4. `dereferencing pointer to incomplete type`

   Este error aparece cuando se utiliza un puntero a una estructura de datos para acceder a alguno de sus campos, pero el compilador no tiene información suficiente sobre esa estructura de datos. El siguiente programa muestra esta situación:

```
struct btree * data;
int main (void)
{
  data->size = 0;  /* Información incompleta */
  return 0;
}
```

   La declaración de la variable `data` como puntero a la estructura `btree` es correcta. Para declarar un puntero a una estructura no se necesita la definición de la estructura. Pero en la primera línea de la función `main` ese puntero se utiliza para acceder a uno de los campos de esta estructura. Al no tener esa definición, el error se muestra en pantalla.

   La causa más probable de este error es que la definición de la estructura está en otro fichero, y debe estar presente antes de que un puntero se utilice para acceder a sus campos. Por este motivo, las definiciones de las estructuras se suelen agrupar en ficheros con extensión `*.h` que son incluidos en los ficheros de código.

5. `warning: control reaches end of non-void function`

   Esta advertencia aparece cuando una función se ha declarado como que devuelve un resultado y no se incluye ningún comando `return` para devolver ese resultado. Por tanto, o se ha definido incorrectamente la función, o se ha olvidado este comando. El compilador igualmente genera el ejecutable y aunque la función no devuelva resultado, el compilador sí devuelve un resultado a la función que ha hecho la invocación.

6. `warning: unused variable `...'`

   Esta advertencia la imprime el compilador cuando detecta que una variable ha sido declarada

pero no se utiliza en ningún sitio. El mensaje desaparece borrando la declaración.

7. `undefined reference to `...'`

   Este mensaje aparece cuando en el código se ha invocado una función que no está definida en ningún lugar. El compilador nos dice que hay una referencia a una función sin definición. Mira qué función falta y cerciórate de que su definición se compila.

8. `error: conflicting types for `...'`

   Se han encontrado dos definiciones del prototipo de una función. Una es un prototipo (tipo del resultado, nombre, paréntesis con los parámetros y un punto y coma) y la otra es la definición con el código. La información en ambos lugares no es idéntica y por tanto existe un conflicto. El compilador te muestra en qué linea se ha encontrado el conflicto y la definición previa que ha causado la contradicción.

12.12.3. Error en la ejecución

La ejecución de programas en C puede terminar abruptamente cuando se detecta un error. Pero a diferencia de otros lenguajes de programación, cuando se produce un error, en lugar de imprimir un informe detallado sobre el lugar en el que se produjo la anomalía, los programas en C imprimen el escueto mensaje **Segmentation fault**. No hay información adicional sobre qué ha causado el error, y debes revisar el programa y su ejecución para corregirlo.

Bibliografía

[Gough05] Brian J. Gough. *An Introduction to GCC*. Network Theory Ltd. Copyright © 1995 Network Theory Ltd. Capítulo 13. *Common Error Messages* .