Compilación de archivos C

Sistemas Operativos 1

24 de Marzo del 2018

Índice

1. Introducción

En este documento nos centraremos en temas de compilación y generación de ejecutables en C. En C, tenemos distintos tipos de archivos: archivos de código (`.c`), archivos de cabecera (`.h`), archivos object (`.o`), librerías (`.a` y `.so`) y archivos ejecutables. Trabajaremos en el proceso de generación de un archivo ejecutable en C a partir de varios archivos de código fuente con un cierto nivel de dependencia. Veremos cómo se puede utilizar en C parámetros pasados por línea de comandos. Trabajaremos con la creación y uso de librerías dinámicas y estáticas y veremos la diferencia entre ambas. Finalmente, aprenderemos cómo hacer un `Makefile`.

2. Compilación de archivos C

Junto con este documento encontrarás 3 archivos de código C (`main.c`, `calculator.c` y `counter.c`) y 2 archivos header (`calculator.h` y `counter.h`). Analiza el contenido de los archivos. Observa las funciones que se llaman desde la función `main.c`, dónde están declaradas y dónde se encuentra su código fuente.

2.1. Archivos en C

En C, los archivos header (`.h`) se utilizan para declarar variables o funciones, mientras que el código fuente se encuentra en un archivo de código aparte (`.c`). Esto facilita la mantención y reutilización del código. Los compiladores de C no permiten utilizar una función a no ser que haya sido previamente declarada. Si una función es utilizada en varios sitios, en lugar de repetir la declaración de la función en cada archivo, se define en un solo lugar (`.h`). Por tanto, para utilizar

una función externa, es necesario hacer primeramente un `include` del archivo header donde se encuentra la declaración de esta función.

2.2. Compilación manual

En C, el proceso de generar un ejecutable a partir de código fuente consta de varios pasos. Primeramente el código es preprocesado. El preproceso es un programa que se ejecuta de manera automática cuando antes de realizar realmente la compilación. Lo que hace el preproceso es modificar o expandir el código fuente. Los comandos de preproceso comienzan con el signo "#". Dos de los más importantes son:

- `#define`: se utiliza principalmente para definir constantes, por ejemplo:

  ```
  #define BIGNUM 1000
  ```

  especifica la cadena BIGNUM vale 1000. El preprocesador, donde quiera que se encuentre BIGNUM en el código, lo sustituirá por 1000.

- `#include`: se utiliza para acceder a la declaración de funciones definidas en otro archivo. Por ejemplo:

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

  hace que el preprocesador copie el contenido del fichero `stdio.h` en el código, en la posición donde se encuentra el `include`.

Luego que el código es preprocesado (y expandido), se realiza la compilación, la cual genera a partir del archivo de código fuente (`.c`) un archivo object (`.o`), el cual contiene una versión binaria del código. Para compilar un archivo `.c`, se utiliza el siguiente comando:

```
>> gcc -c filename.c -o filename.o
```

Los archivos object no son directamente ejecutables, ya que hay que añadir especificaciones para las funciones y librerías externas que han sido incluidas. El *linker* (enlazador) hace este trabajo, y a partir de varios ficheros `.o` genera un ejecutable. Para generar un ejecutable a partir de varios archivos object (`.o`), se utiliza el siguiente comando:

```
>> gcc file1.o file2.o file3.o -o myprog
```

Problema 1 *A partir del código del documento, abre la consola y ve a la carpeta donde se encuentran los archivos del documento y realiza los siguientes pasos:*

1. *Compila los archivos de código C por separado y genera los archivos object correspondientes (`main.o`, `calculator.o` y `counter.o`)*
2. *Haz el* link *de los archivos object `main.o`, `calculator.o` y `counter.o` y genera un ejecutable llamado `myprog`. Como en calculator se utiliza la librería `<math.h>` será necesario añadir la opción `-lm` al final del comando.*
3. *Ejecuta `myprog` (con `./myprog`) y comprueba que el programa se ejecuta correctamente*
4. *Observa qué ocurre si eliminas `calculator.o` al generar el ejecutable. ¿Qué tipo de error da? ¿Por qué ha ocurrido este error, si no hemos modificado el código fuente y todos los `include` están correctos?*

2.3. Librerías estáticas

Las librerías estáticas encapsulan un conjunto de funciones y variables. Cuando se incluyen librerías estáticas en el código, estas son cargadas en tiempo de compilación y son copiadas dentro del ejecutable final. La ventaja de las librerías estáticas radica en la certeza de que el ejecutable final contiene dentro todas las funciones necesarias para su ejecución. Es posible producir un único fichero final, lo cual simplifica su distribución e instalación. Por convención, el nombre de la librería siempre comienza por `lib` y tiene una extensión `.a`.

Para crear una librería estática en C a partir de varios ficheros `.o` utilizamos el siguiente comando:

```
>> ar rcs libname.a file1.o file2.o file3.o
```

Para utilizar una librería estática, haremos el `include` correspondiente a sus funciones y a la hora de generar el ejecutable lo haremos de la siguiente manera:

```
>> gcc file4.o file5.o -o myprog libname.a
```

Problema 2 *A partir de lo visto en esta sección, genera una librería estática a partir de los ficheros `calculator.c` y `counter.c` y realiza el enlace con el fichero objeto de `main.c`.*

2.4. Librerías dinámicas

Las librerías dinámicas también encapsulan un conjunto de funciones y variables. La mayor diferencia radica en que estas librerías no son incluidas dentro del ejecutable sino que son cargadas en memoria en tiempo de ejecución. Debido a ello, deben ser distribuidas junto con el archivo ejecutable. La ventaja de tener librerías dinámicas radica en que su código puede ser actualizado y recompilado sin necesidad de modificar los ejecutables que la utilizan, haciendo así más fácil la mantención. Por convención, el nombre de una librería dinámica comienza por `lib` y tiene una extensión `.so` (shared object). Para crear una librería dinámica en C a partir de varios ficheros `.o` debemos:

- Generar los ficheros `.o` con la opción `-fPIC`

  ```
  >> gcc -c -fPIC file1.c file2.c file3.c
  ```

- Crear la librería a partir de los ficheros `.o` con el siguiente comando:

  ```
  >> gcc -shared -o libname.so file1.o file2.o file3.o
  ```

  Para utilizar una librería dinámica, haremos el `include` correspondiente a sus funciones y a la hora de generar el ejecutable lo haremos de la siguiente manera:

  ```
  >> gcc file4.o file5.o  -o myprog ./libname.so
  ```

  Observar que para poder compilar hay que especificar el camino (*path) a la librería dinámica a utilizar.*

Problema 3 *A partir de lo visto en esta sección, genera una librería dinámica a partir de los ficheros* `calculator.c` *y* `counter.c` *y realiza el enlace con el fichero objeto de* `main.c`*.*

Problema 4 *Una vez realizados los problemas anteriores, imagina que se hace únicamente una modificación en el código de* `calculator.c`*. ? Qué debemos hacer para que esta modificación sea tenida en cuenta cuando se ejecute* `myprog_static` *y* `myprog_dynamic`*? Piensa en la solución mas eficiente en cada caso (*`myprog_static` *y* `myprog_dynamic`*).*

2.5. Archivos Makefile

La utilidad `make` se utiliza habitualmente para poder compilar archivos en cualquier lenguaje. En vez de invocar a `gcc` directamente, se invoca el comando `make`. Este comando lee el fichero `Makefile` que incluye las reglas para poder compilar un programa. La idea es simplificar el proceso de generación del ejecutable y asegurarse de que solo el código que ha sido modificado es recompilado. Esto no es tan importante cuando los archivos de código fuente son pocos, pero para grandes proyectos con cientos de archivos de código rápidamente se convierte en un problema! El `Makefile` es similar a un lenguaje de programación delcarativo: se establecen las relaciones entre los comandos a ejecutar pero no se establece realmente el orden de ejecución. Aquí sólo veremos algunos ejemplos sencillos de lo que se puede hacer. Para más información puedes ver: https://www.gnu.org/software/make/manual/

El archivo `Makefile` contiene reglas, las cuales están compuestas de:

- Un *target*: Usualmente el nombre de un archivo, también puede ser el nombre de una acción, por ejemplo *clean* para eliminar archivos
- Una dependencia: Es otro *target* que tiene que ser procesado antes que el target actual, normalmente son archivos que son necesarios para crear el target actual
- Un comando: Es una acción que se debe ejecutar. Puede consistir de uno o más comandos, cada uno en una línea. Importante: debes hacer un tab al inicio de cada línea de comando en los archivos `Makefile`, omitir este paso es un error muy frecuente

Problema 5 *Abre el archivo* `Makefile` *y analiza su contenido. ¿Eres capaz de encontrar el target, las dependencias y los comandos en este archivo? Rellena las partes que faltan, de modo que el* `Makefile` *realice la misma funcionalidad que el Problema (1). Comprueba el resultado ejecutando el comando* `make` *en la consola.*

Prueba de modificar cualquier fichero C y observa que al ejecutar `make` únicamente se compilan aquellos ficheros que realmente son necesarios. El comando `make` es el que determina los ficheros a compilar a partir de las dependencias establecias en el fichero `Makefile`.

Coje a continuación el archivo `Makefile2` y sobreescribe el `Makefile` que has generado. Observa que este nuevo `Makefile` genera la libreria estática. Además, utiliza las denominadas reglas implícitas para poder compilar los ficheros C en ficheros objeto. Las reglas implícitas son últiles para establecer cómo se "transforman"ficheros.