Programación Avanzada

Instructivo de Compilación

1 Índice

1 ÍNDICE 2

2 INTRODUCCIÓN 3

3 HERRAMIENTAS 3

3.1 SINTAXIS DE LOS COMANDOS GCC Y G++ 3
3.2 OTRAS HERRAMIENTAS: 4
3.3 EJEMPLOS: 5

***4* EL COMANDO *MAKE* 5**

4.1 EJEMPLOS 6

5 REFERENCIAS 10

2 Introducción

Este instructivo introduce en la compilación y linkedición de programas desarrollados en C o C++, utilizando compiladores GNU. Por más información referirse a las páginas *man* de los compiladores.

3 Herramientas

Al compilador de C se accede mediante el ejecutable "*gcc*", y al compilador C++ mediante el ejecutable "*g++*".

3.1 Sintaxis de los comandos gcc y g++

A continuación se presenta una versión restringida de la sintaxis del comando `gcc`, la cual es análoga a la del `g++`.

```
gcc [-c] [opciones] [-o nombreOutPut] nombreInput
```

Por defecto si no se especifican opciones `gcc` compila y hace la linkedición de los archivos, generando como resultado un ejecutable de nombre *`a.out`* (en general, sin embargo, se usa la opción `–o` para especificar el nombre del archivo de salida). En el caso que se use la opción `–c`, `gcc` solamente compila, es decir genera solamente código objeto (`.o`). La opción `–o` permite especificar el nombre del archivo de salida del compilador (este puede ser tanto el nombre del ejecutable como del archivo objeto).

Otras opciones:

- `-Wall` hace que el compilador genere ***warnings*** ante cualquier detalle posiblemente incorrecto del programa. Útil en la etapa de desarrollo para detectar ***bugs*** y en general para programar con un mejor estilo.

- `-g` hace que el compilador deje en el código objeto información acerca de las líneas de código del programa. Necesario para ejecutar el programa con un depurador como `gdb` o `ddd`.

- `-lstdc++` incluye la biblioteca estándar de C++ para linkeditar el programa. No es necesario en las salas de máquinas pues está incluida por defecto, pero puede ser necesaria en otros ambientes según la configuración de los mismos.

3.2 Otras herramientas:

A continuación se presenta una lista de herramientas que pueden resultar útiles para el desarrollo en C++.

3.2.1 DISPONIBLES EN LAS SALAS DE MÁQUINAS:

- gdb (GNU DebuGger)

Depurador básico de línea de comandos. Permite ejecutar un programa línea a línea, examinar el valor de las variables en tiempo de ejecución, etc. Es necesario compilar y linkeditar el programa con la opción –g.

Modo de ejecución:
`gdb ./`***programa***

- dos2unix

Transforma un archivo de texto de formato DOS a formato Unix, eliminando los caracteres ^M que aparecen cuando se edita un archivo de texto con formato DOS en Unix o Linux.

Modo de ejecución:

`dos2unix` ***"archivo origen" "archivo destino"***

- vi

vi es el editor clásico de los sistemas Unix. Está disponible en prácticamente cualquier sistema Unix.

- emacs

Emacs es un editor de texto de terminal extensible y customizable, muy utilizado para programar.

- Eclipse

Entorno de desarrollo para múltiples lenguajes, que en particular posee herramientas para el desarrollo en C/C++. El editor tiene entre otras características: resaltamiento de sintaxis, asistencia al codificar (brindando sugerencias), compilación automática al modificar un fuente (con reporte de errores amigable) y debugger incorporado en el entorno.

3.2.2 NO DISPONIBLES EN LAS SALAS DE MÁQUINAS

- Anjuta

Anjuta es un entorno de desarrollo para C/C++ sobre GNOME bastante avanzado. Incluye varias herramientas y utilidades orientadas a facilitar el trabajo de los desarrolladores.

- xemacs

Editor de código muy completo y customizable, con resaltamiento de sintaxis, tabulación inteligente, etc..

- ddd

Depurador gráfico, más amigable que el gdb. Tiene funcionalidades análogas a las del gdb. También es necesario compilar y linkeditar el programa con la opción –g

3.3 Ejemplos:

Ejemplo	Descripción
`gcc –o prueba prueba.c`	Compila el archivo C `prueba.c` y realiza el link. Si no se producen errores el resultado es un ejecutable con el nombre prueba.
`g++ -o prueba prueba.cc`	Idem al anterior pero compilado con el compilador C++. Observar que el sufijo es `.cc`.
`g++ -c prueba.cc`	Solamente compila el archivo C++ `prueba.cc`. El resultado es un archivo binario `prueba.o`
`g++ –o prueba prueba.cc module1.o module2.o`	Compila el archivo en C++ `prueba.cc`, hace el link con los binarios `module1.o` y `module2.o`

4 El comando *make*

El comando *make* típicamente es utilizado para simplificar el proceso de compilación y linkedición de un programa compuesto por varios archivos (los cuales pueden ser pensados como módulos). El *make* por defecto toma como entrada un archivo con el nombre "*Makefile*" situado en el directorio. Este archivo describe los pasos que se debe seguir para generar el programa.

El archivo *Makefile* establece reglas de dependencias (normalmente entre archivos) y comandos que se deben ejecutar cuando esas dependencias se cumplen. En principio las dependencias y comandos pueden ser cualquiera, aunque en general se utilizan dependencias entre archivos de código y comandos de compilación.

Un *Makefile* consiste entonces de una secuencia de parejas regla-acción. El comando *make* lee estas parejas, decide si la dependencia se cumple (puede ser

que no haya dependencias, en cuyo caso siempre se ejecuta la acción) y en ese caso ejecuta la acción especificada (es posible especificar más de una acción).

Una dependencia entre archivos se cumple cuando el archivo dado ha sido modificado desde la última vez que se ejecutó *make*.
Un *Makefile* también permite utilizar variables, reglas condicionales, etc. como todo ***script.***

Formato general de un *Makefile*:

```
regla1: dependencia
<tabulador> acción
<tabulador> acción
…

reglan: dependencia
<tabulador> acción
<tabulador> acción
```

donde
dependencia = archivos de que depende la regla 1
acción = acción a ejecutar cuando los archivos especificados fueron modificados

En cualquier punto se pueden declarar o utilizar variables, según la sintaxis del ***shell*** utilizado. En los ejemplos se declaran y utilizan variables con una sintaxis que funciona en el ***shell*** por defecto de los usuarios Linux de las salas de máquinas.

Un error común es olvidar el tabulador, que es necesario para el correcto procesamiento del *Makefile*.

4.1 Ejemplos

- Ejemplo 1

```
ejec: ejec.cc ejec.h modulo1.o modulo2.o modulo3.o
        g++ -o ejec ejec.cc modulo1.o modulo2.o modulo3.o

modulo1.o: modulo1.cc modulo1.h modulo3.o
        g++ -c -o modulo1.o modulo1.cc

modulo2.o: modulo2.cc modulo2.h modulo3.o
        g++ -c -o modulo2.o modulo2.cc

modulo3.o: modulo3.cc modulo3.h
        g++ -c -o modulo3.o modulo3.cc
```

En este ejemplo, `ejec` depende de los archivos `ejec.cc`, `ejec.h`, `modulo1.o`, `modulo2.o` y `modulo3.o`. El comando *make* resuelve recursivamente las dependencias. Resolver la dependencia de `ejec` significa que si alguno de los archivos dependientes cambia, se debe regenerar `ejec`, ejecutando:

```
g++ -o ejec ejec.cc modulo1.o modulo2.o modulo3.o
```

Por ejemplo, si luego de la última vez que se ejecutó *make* se modificó el archivo `modulo3.h`, al ejecutar *make* éste detecta que es necesario aplicar las siguientes reglas:

- `g++ -c -o modulo3.o modulo3.cc`

Para regenerar el archivo `modulo3.o`, dado que uno de los archivos de los que depende (`modulo3.h`) ha sido modificado.

- `g++ -c -o modulo2.o modulo2.cc`

Para regenerar el archivo `modulo2.o`, dado que uno de los archivos de los que depende (`modulo3.o`) ha sido modificado (fue recompilado en el paso anterior).

- `g++ -c -o modulo1.o modulo1.cc`

Para regenerar el archivo `modulo1.o`, dado que uno de los archivos de los que depende (`modulo3.o`) ha sido modificado (fue recompilado en un paso anterior).

- `g++ -o ejec ejec.cc modulo1.o modulo2.o modulo3.o`

Para regenerar el archivo `ejec`, ya que algunos de los archivos de los que depende han sido modificados (todos los `.o`).

- Ejemplo 2

```
OPCIONES = -g

CC = g++

ejec: ejec.h ejec.cc
     $(CC) –o ejec ejec.cc $(OPCIONES)
```

En este ejemplo se declaran dos variables (`OPCIONES` y `CC`). Luego al generar el ejecutable se resuelven estas variables sustituyéndolas por el valor declarado y entonces la línea de ejecución utilizada es: `g++ –o ejec ejec.cc –g`.

Utilizar variables permite simplificar las reglas de dependencia y comandos utilizados, haciéndolos más compactos y menos propicios a errores. A la vez

permite modificar los parámetros utilizados al compilar de manera simple y uniforme. En este ejemplo, si se desea dejar de utilizar la opción –g para compilar, basta con modificar el valor de la variable OPCIONES.

- Ejemplo 3

```
OBJETOS = modulo1.o modulo2.o

FUENTES = modulo1.cc modulo1.h \
modulo2.cc modulo2.h \
ejec.cc ejec.h

CC = g++
OPCIONES = -g –Wall
SINUSO =

ejec: $(OBJETOS) ejec.h ejec.cc Makefile
     $(CC) $(OPCIONES) $(OBJETOS) $(SINUSO) ejec.cc -o ejec

modulo1.o: modulo1.h modulo1.cc
     $(CC) $(OPCIONES) –c modulo1.cc -o modulo1.o

modulo2.o: modulo2.h modulo2.cc
     $(CC) $(OPCIONES) –c modulo2.cc –o modulo2.o

clean:
     rm –f $(OBJETOS) ejec

rebuild:
     make clean
     make

zip: $(FUENTES) Makefile
     rm –f codigo.tar.gz
     tar –cvf codigo.tar $(FUENTES) Makefile
     gzip codigo.tar
```

Observe en el ejemplo el uso de "\" para colocar el contenido de una variable en más de un renglón.

En este ejemplo, son válidos los siguientes comandos:

- `make modulo1.o`

Si alguno de los archivos `modulo1.h` o `modulo1.cc` ha sido modificado, se ejecuta la regla que en este caso es:

`g++ –g –Wall -c modulo1.cc –o modulo1.o`.

- `make modulo2.o`

Análogo al anterior.

- `make ejec`

Genera el ejecutable `ejec`, resolviendo recursivamente las dependencias. Observar que `ejec` depende del propio *`Makefile`*.

- `make`

Esto ejecuta la acción por defecto del *`Makefile`*, o sea la primera que en este caso es `ejec`. O sea que es equivalente a escribir `make ejec`.

- `make rebuild`

Siempre ejecuta en secuencia `make clean` y `make` (ya que no tiene dependencias).

- `make clean`

Elimina todos los archivos de código objeto declarados en la variable `OBJETOS` y el ejecutable `ejec`.

- `make zip`

Si cualquiera de los archivos de código fuente o el propio *`Makefile`* fue modificado, borra el archivo de código zipeado anterior (si no existe no importa) y genera uno nuevo que incluye todo el código fuente y el *`Makefile`* (útil para generar versiones entregables con todo el código fuente, sin objetos o ejecutables).

5 Referencias

[1]	Make http://www.gnu.org/software/make/manual/make.html man make
[2]	Xemacs www.xemacs.org
[3]	Gcc http://gcc.gnu.org man gcc
[4]	Vi http://www.thomer.com/vi/vi.html http://www.eng.hawaii.edu/Tutor/vi.html man vi
[5]	Ddd http://www.gnu.org/manual/ddd/index.html
[6]	Gdb http://sources.redhat.com/gdb/ man gdb
[7]	Anjuta http://anjuta.sourceforge.net/
[8]	Emacs http://www.gnu.org/software/emacs/manual/emacs.html http://www.emacswiki.org/cgi-bin/emacs-es/WikiEmacs
[9]	Eclipse http://www.eclipse.org/