Compilación y depuramiento de programas C en el sistema operativo Unix

Dr. Roberto Gómez Cárdenas
ITESM-CEM, Dpto. Ciencias Computacionales
rogomez@campus.cem.itesm.mx
http://webdia.cem.itesm.mx/dia/ac/rogomez

6 de febrero de 2002

El presente documento representa un borrador de lo que será el capítulo de un libro. Acaba de ser terminado y debido a la urgencia de tiempo no se presto mucha atención a la forma, pero el fondo es muy bueno. Cualquier aclaración, comentario o correción se agradecera que se le notifique al autor vía personal o a través del correo electrónico. Así mismo vale la pena aclarar que una gran parte de este documento esta basado en el capítulo 13 de [1].

1 Introducción

La mayor parte de los programadores de hoy en día realizan sus programas en lenguajes denominados de alto nivel. Para que una computadora pueda entender las instrucciones del programador es necesario compilar el programa. La compilación tiene dos objetivos: el detectar posibles errores sintácticos en el programa y el generar código que la máquina pueda entender y ejecutar.

Existe una gran variedad de compiladores en el mercado, a nivel PC los compiladores tipo Turbo adquirieron gran popularidad en los años 80's. Sin embargo estos compiladores sólo podian ser usados en máquinas personales con un sistema operativo tipo DOS y Windows. Para sistemas Unix, el compilador por excelencia es el denominado: `cc`,(C-Compiler) el cual no es considerado muy amigable por la gente acostumbrada a manejar compiladores Turbo o Borland. Sin embargo, el compilador ofrece una serie de opciones que muchos usuarios desconocen.

Una vez compilado el programa es necesario verificar que los resultados que ofrece son los que esperamos. Sin embargo si no es el caso, es necesario saber qué fue lo que pasó; en otras palabras, es necesario "depurar" el programa.

El depurar un programa implica el usar cierto tipo de herramientas que todo programador debe conocer. El objetivo de depurar un programa es es conocer todas las operaciones que se están llevando a cabo para obtener un resultado. Lo anterior implica conocer el valor de variables locales, globales y ver el flujo del programa.

Los compiladores tipo Borland/Turbo traen integradas herramientas que permiten realizar una buena depuración de los programas. Mucha gente piensa que la única opción para depurar programas realizados sobre Unix, es desplegar mensajes en puntos críticos del programa. Mensajes estilo "*Antes de hacer x* o "*Despues de hacer y*". Unix proporciona buenas herramientas para poder depurar programas que los principiantes desconocen, como son `gdb`, `dbx` ó `ddd`.

Por último una de las tendencias para el diseño e implementación de sistemas es lo que se conoce como modularidad. Lo anterior consiste en construir módulos independientes de códigos que al final van a formar todo el sistema. Unix proporciona una buena herramienta para definir las relaciones entre estos módulos y no tener que compilar módulos previamente compilados. La utilería se conoce como `make`.

Este documento tiene por objetivo el dar a conocer las características principales del compilador `cc`. También se dan a conocer los aspectos principales de los depuradores `gdb`, `ddd` y `dbx`. Por último se presentan los aspectos principales de la utilería `make`.

2 Compilando un programa

Cuando uno necesita desarrollar una aplicación en ambiente Unix, lo primero es escribir el código de programa usando un editor. Existen varios editores para hacer lo anterior. Entre los más recomendados podemos mencionar vi, emac, textedit, etc. No es intención del presente documento el presentar dichos editores.

Una vez que se cuenta con el programa, el siguiente paso es compilarlo. El compilador `cc`[1] cuenta con la siguiente sintaxis:

```
cc <progama> [opciones]
```

donde `<programa>` es el nombre del programa a compilar y debe contar con una extensión `.c`. Por ejemplo, para compilar el programa `toto.c`, se teclea lo siguiente:

```
$ cc toto.c
```

La utilidad `cc` llama al preprocesador de C, al compilador de C, al ensamblador y al ligador. El preprocesador de C expande constantes simbólicas y definiciones de macro y también incluye archivos de encabezado (en general todo lo que es precedido por un `#`). La fase de compilación crea el código de lenguaje ensamblador correspondiente a las instrucciones en el archivo fuente. Luego el ensamblador crea un código objeto. Un archivo objeto es creado por cada archivo fuente. Cada archivo objeto tiene el mismo nombre que el archivo fuente, con la excepción de que la extensión `.c` es reemplazada con un `.o`. En el ejemplo anterior, un solo archivo objeto será creado, `toto.o`. Como sea, después de que el compilador de C completa exitosamente todas las fases de compilación para un programa contenido en un solo archivo fuente, crea el archivo ejecutable y después borra el archivo `.o`. Si se compila exitosamente `toto.c`, no se verá el `toto.o`.

El nombre del archivo ejecutable por default es `a.out`. Basta con escribir este archivo en el shell para ejecutar lo que se compiló.

A nivel de ejemplo presentamos el siguiente código:

```
int main(void)
{
   printf("Hola mundo \n");
   return 0;
}
```

[1] El compilador `cc` viene con algunas distribuciones Unix propietarias y hay que pagar por él, la versión libre de `cc` es la producida por GNU: `gcc`. Todo lo explicado en este documento con respecto a `cc` se aplica de igual manera a `gcc`.

La compilación del código anterior da como resultado:

```
$ ls
toto.c
$ cc toto.c
$ ls
a.out* toto.c
$ a.out
Hola mundo
$
```

El comando `ls` de Unix despliega el contenido del directorio. El `*` al lado del archivo `a.out` significa que el archivo es ejecutable.

Para ejecutar un comando, o un programa, el sistema operativo busca en una variable de ambiente las diferentes rutas (path) donde puede encontrarse el archivo ejecutable. La mayor parte de los sistemas Unix incluyen el directorio de trabajo, sin embargo Linux no.

Para ejecutar el programa `a.out`, los usuarios de linux tendrían que teclear `.\` antes del archivo ejecutable. En el caso anterior:

```
$ ./a.out
Hola mundo
$
```

2.1 La opción -l y el ligador

Durante la fase final del proceso de compilación, el ligador busca bibliotecas específicas para funciones que usa el programa en cuestión y combina módulos objeto para esas funciones con los módulos objeto del programa. Por default, el compilador busca la biblioteca estándar de C `libc.a`, que contiene funciones que se encargan de las operaciones de entrada y salida y provee otras muchas capacidades de propósito general. Si se quiere usar el ligador para buscar otras bibliotecas, se debe usar la opción `-l` para especificar las bibliotecas en la línea de comando. A diferencia de la mayoría de las opciones para utilidades en sistemas UNIX, la opción `-l` no viene antes de todos los nombres de archivos en la línea de comandos, sino que viene después de todos los nombres de archivos de todos los módulos que se aplican. En el siguiente ejemplo, el compilador busca la librería math, `libm.a`:

```
$ cc calc.c -lm
```

Como se puede ver del ejemplo, la opción `-l` usa abreviaciones para los nombres de las bibliotecas, anexando la siguiente letra `-l` a `lib` y agregando una extensión `.a`. La `m` en el ejemplo quiere decir `libm.a`.

Es necesario saber qué tipo de liga se debe aplicar al compilar un programa. Lo anterior depende de las funciones usadas dentro de un programa. Sin embargo el manual de unix (`man`) puede servirnos para saber cómo ligar un programa.

Para ejemplificar lo anterior presentamos el siguiente código:

```
#include <math.h>
```

```
int main(void)
{
   float x;
   int y;

   y = 20;
   x = sqrt(y);
   printf("La raiz cuadrada de %d es %f \n",y,x);
   return 0;
}
```

El compilar el programa anterior, sin las ligas apropiadas, da como resultado:

```
$ cc raiz.c
 /var/tmp/ccxRaaAR1.o: In function ‘main’:
 /var/tmp/ccxRaaAR1.o(.text+0x24): undefined reference to ‘sqrt’
$
```

A partir de la salida anterior podemos apreciar que el problema está en la función `sqrt()`. Para poder conocer cómo se debe compilar utilizamos el comando `man`.

```
$ man sqrt
Reformatting page.  Wait... done

Mathematical Library                                    sqrt(3M)

NAME
     sqrt - square root function

SYNOPSIS
     cc [ flag ... ] file ... -lm [ library ... ]
     #include <math.h>

     double sqrt(double x);

DESCRIPTION
          :
          :
```

La sección de SYNOPSIS nos indica que es necesario compilar el programa con la opción `-lm`. Compilando y ejecutando el programa:

```
$ cc raiz.c -lm
$ a.out
La raiz cuadrada de 20 es 4.472136
$
```

Dependiendo la función será la liga necesaria. Desafortunadamente dependemos de que los manuales esten cargados en el sistema y de que la función que se busca este dada de alta.

2.2 Opción -o: cambiando el nombre del archivo ejecutable

El nombre por default del archivo ejecutable es `a.out`. Si se compila un programa diferente al de la última compilación, el compilador sobreescribirá el nombre del archivo, sin preguntar nada, con el mismo nombre de default: `a.out`.

Una opción para guardar los archivos ejecutables producidos por diferentes programas es cambiar el nombre por default a uno más descriptivo.

```
$ mv a.out toto
```

Otra opción más eficiente es especificar el nombre del ejecutable cuando se usa `cc`. Si se usa la opción `-o`, el compilador dará al ejecutable el nombre de elección en lugar de `a.out`.

Tomando como base el ejemplo de la sección anterior, si deseamos que el archivo ejecuutable se llame `raiz`, debemos indicarselo con la opción `-o`. Para ejecutar el programa basta con teclear el nombre del archivo ejecutable.

```
$ ls
raiz.c
$ cc raiz.c -o raiz
$ ls
raiz* raiz.c
$ raiz
La raiz cuadrada de 20 es 4.472136
$
```

2.3 Compilando varios módulos

Una de las técnicas más recomendadas para el diseño e implementación de grandes sistemas es el programar por módulos. Es decir, contar con diferentes archivos cada uno de los cuales contiene código que forma parte de todo un sistema.

Si se quiere compilar algunos pero no todos los módulos de un programa, se puede usar la opción `-c` en `cc`, que suprime la fase de enlazar ligas. La opción `-c` es útil porque no trata con referencias externas sin resolver como errores, esta capacidad permite compilar y depurar la sintaxis de los módulos de un programa conforme se van creando. Una vez que se han compilado y depurado todos los módulos, se puede correr `cc` otra vez con los archivos objeto como argumentos para producir un programa ejecutable. En el siguiente ejemplo, `cc` produce tres archivos objeto pero no ejecutables:

```
$ cc -c ledger.c  acctsrec.c  acctsrec.c
$ ls
acctspay.c  acctspay.o  acctsrec.c  acctsrec.o  ledger.c  ledgar.o
$
```

Si se corre `cc` otra vez usando los archivos objetos, `cc` producirá el ejecutable. Debido a que el compilador reconoce la extención del nombre del archivo `.o`, reconoce que los archivos solo necesitan ser ligados. También se puede incluir los archivos `.c` y `.o` en una sola línea de comandos, como en el siguiente ejemplo:

```
$ cc -o accounting ledger.o  acctspay.c  acctsrec.o
```

El compilador reconoce que el archivo `.c` necesita ser preprocesado y compilado, mientras que el `.o` no. También acepta archivos de lenguaje ensamblador que terminan con `.s`, y los trata apropiadamente (esto es, `cc` los ensambla y los liga). Esta característica hace fácil de modificar y recompilar un programa.

Con programas largos que tienen muchos módulos, la habilidad para compilar módulos por separado y ligarlos con módulos previamente compilados puede ahorrar mucho tiempo. La utilería make provee un método automático para saber qué módulos necesitan ser recompilados. Las características de esta utilería serán descritos en la sección 7.

2.4 Opción -O: el optimizador de código

La opción `-O` causa que `cc` use el optimizador del compilador. El optimizador crea código objeto más eficientemente, de tal manera que el programa ejecutable corra más rápido.

```
$ cc -O ledger.c acctspay.c acctsrec.c
$ ls
a.out acctspay.o acctsrec.o ledger.o
acctspay.c acctsrec.c ledger.c
```

2.5 Otras opciones

Recordemos que `cc` es un compilador hecho para una determinada plataforma. Dependiendo del tipo de compilador `cc` usado, se cuentan con otras opciones para diferentes usos. Si el lector desea conocer todas se le recomienda usar el comando `man`.

Una opción interesante, y que está ligada con la siguiente sección, es la opción `-g` que produce información de depuración para el sistema operativo.

3 Depurando programas escritos en C sobre Unix

El compilador de C es libre para los tipos de construcciones que permite en programas. Como otras muchas utilerías de UNIX, `cc` parece estar basado en la filosofía de que el usuario realmente quiere decir lo que dice y que no tener noticias son buenas noticias. El compilador permite casi todo lo que lógicamente posible de acuerdo con la definición del lenguaje. A pesar de que esto le da al programador gran flexibilidad y control, puede hacer también la corrección de errores difícil.

Un camino para la corrección de errores en un programa en C es desplegar mensajes en puntos críticos del código fuente.

Considere el siguiente ejemplo:

```
int main(void)
{
  int a,b;
```

```
  printf("Programa de prueba \n");
  a = 0;
  b = 459 + a;
  a = a/0;
  b = b * a;
  printf("Valores finales a y b: %d, %d \n",a,b);
  retunr 0;
}
```

Al compilarlo y ejecutarlo obtenemos:

```
$ cc cachafas.c -o cachafas
$ cachafas
Programa de prueba
Arithmetic Exception(coredump)
$
```

Cuando un programa tiene algún error, el sistema operativo no nos indica en qué línea ocurre el error. Así mismo el mensaje de error no es muy explícito tampoco. Una posible solución para encontrar el error, es desplegar mensajes antes de cada asignación. Regresando a nuestro ejemplo este quedaría de la siguiente forma:

```
int main(void)
{
  int a,b;

  printf("Programa de prueba \n");
  a = 0;
  printf("Despues 1a. asignacion \n");
  b = 459 + a;
  printf("Despues 2da. asignacion \n");
  a = a/0;
  printf("Despues 3a. asignacion \n");
  b = b * a;
  printf("Valores finales a y b: %d, %d \n",a,b);
  return 0;
}
```

Volviendo a compilar y ejecutar:

```
$ cc cachafas.c -o cachafas
$ cachafas
Programa de prueba
Despues 1a. asignacion
Despues 2da. asignacion
Arithmetic Exception(coredump)
$
```

Por lo que asumimos que el error esta despues de la segunda asignación, y antes de la tercera.

Sin embargo hay que tener cuidado en que todos las funciones `printf()` tengan un salto de línea (caracter `\n`). En caso contario se puede causar confusiones. Si en el ejemplo anterior cambiamos el mensaje `Despues 2da. asignacion` por:

```
printf("Despues 2da. asignacion ");
```

y después compilamos y ejecutamos:

```
$ cc cachafas.c -o cachafas
$ cachafas
Programa de prueba
Despues 1a. asignacion
Arithmetic Exception(coredump)
$
```

La salida anterior nos da a entender que el error esta después de la primera asignación y no como en realidad es. De ahí la importancia de utilizar saltos de línea[2].

Otra opción es utilizar la instrucción `fprintf()` en lugar de `printf()`. La función `fprintf()` envía sus mensajes a la salida de errores estándar, así que si se redirige la salida de este programa se pueden separar de la salida estándar.

Para poder usar `fprintf()` es necesario usar el archivo de encabezado `stdio.h`.

El código con el que se ha estado trabajando se tendría que modificar a:

```
#include <stdio.h>
int main(void)
{

  int a,b;

  printf("Programa de prueba \n");
  a = 0;
  fprintf(stderr,"Despues 1a. asignacion \n");
  b = 459 + a;
  fprintf(stderr,"Despues 2da. asignacioni \n");
  a = a/0;
  fprintf(stderr,"Despues 3a. asignacion \n");
  b = b * a;
  printf("Valores finales a y b: %d, %d \n",a,b);
  return 0;
}
```

La compilación es la misma, sin embargo al ejecutar habría que rederigir la salida de errores estándar a algún archivo.

```
$ cachafas 2> errores
Programa de prueba
Arithmetic Exception(coredump)
$ more errores
Despues 1a. asignacion
Despues 2da. asignacion
$
```

[2] Esto se debe a que $\backslash n$ realiza un flush de stdout.

Como último comentario, vale la pena aclarar que la forma de redireccionar la salida de errores estándar depende del shell que se esté usando. El ejemplo anterior se corrió sobre korn shell.

3.1 Usando lint para encontrar errores en un programa

Para programas sencillos, o en casos en donde se puede tener alguna idea de que está mal en un programa, incluir mensajes en el código puede ayudar a la corrección de errores. El sistema UNIX provee varias herramientas que ayudan a la corrección de errores.

El verificador de programas de C, `lint`, es una de las herramientas más útiles. Checa programas de errores potenciales y problemas de portabilidad. A diferencia del compilador `cc`, `lint` es muy estricto. Detecta y reporta una amplia variedad de problemas y problemas potenciales, incluyendo variables que son usadas antes de contar con algún valor, argumentos para funciones que no son usados y funciones que usan valores de regreso que nunca se regresan.

Consideremos el siguiente código (extraído de [1]):

```
#include <stdio.h>
#define TABSIZE   8

int main(void)
{
   int c, inc, posn = 0;

   while ((c = getchar()) != EOF)
   switch (c)
   {
      case '\t':
        inc = findstop(posn):
        posn += inc;
        for ( ; inc > 0; inc --)
          putchar(' ');
        break;
      case '\n':
        putchar(c);
        posn = 0;
        break;
      default:
        putchar(c);
        posn += inc;
        break;
   }
}

findstop(col)
  int col;
{
  return (TABSIZE - (col % TABSIZE));
}
```

La utilidad `lint` descubre dos problemas con el ejemplo del programa `tabs.c`

```
$ lint tabs.c
tabs.c(27):  warning: main( ) returns value to invocation environment
putc returns value which is always ignored
$
```

Por convención, si un programa corre exitosamente debe regresar un valor cero, si no lo hace, el código de salida no se define. Si se agrega el siguiente enunciado al final de la función `main()` en `tabs.c`, el primer warning desde `lint` puede desaparecer:

```
return 0;
```

El segundo warning desde `lint` podría parecer raro, ya que el programa no incluye ninguna llamada to `putc`. El mensaje hace referencia a `putc` porque `putchar` es una macro que llama a `putc`; el problema con `tabs.c` es que los códigos que regresa de las llamadas a `putchar` nunca son checadas. Si `putchar` falla, el error no será detectado y la salida del programa `tabs` será incorrecta.

A pesar de la libertad de ignorar los warnings de `lint` y seguir compilando el programa, un warning normalmente significa que el programa tiene un error o una construcción no portable, o que se ha violado un estándar de buena programación. Poniendo atención a los warnings de `lint` es un buen camino de corregir un programa y de mejorar habilidades de programación.

3.2 Depuradores de programas

Los sistemas UNIX también cuentan con depuradores para problemas que evaden `lint` y el compilador de C. Entre estos depuradores podemos encontrar `gdb`,`ddd`, `abs`, `sdb` y `dbx`.

Las herramientas `gdb` y `dbx` son depuradores de alto nivel, capaces de analizar la ejecución de un programa en términos de las declaraciones del lenguaje C. También proporcionan una visión de bajo nivel para analizar la ejecución de un programa en términos de las instrucciones de la máquina.

En este documento se analizan tres depuradores: el `dbx`, el `gdb` y el `ddd`. Se les dedica una sección a cada uno.

4 El depurador gdb

GDB es un depurador que, como cualquier otro depurador, permite ver lo que pasa en la memoria de la computadora: detener el programa donde se quiera, ver variables, funciones o regiones de memoria, etc. GDB también permite analizar archivos `core` y depurar un programa que ya ha sido empezado. GDB puede ser usado con lenguajes como `C`, `C++` y `Modula2`.

GDB no tiene una interfaz agradable para XWindows, pero tiene interacción muy buena con gemacs; si se es usuario de gemacs, entonces GDB probablemente será un depurador muy bueno.

Como cualquier software de Gnu, GDB es gratis, protegido por la Licencia General Pública de GNU. Básicamente, la licencia permite copiar o modificar una copia del programa; para los detalles sobre la licencia se recomienda entrar al sitio de Gnu [2].

El propósito de un depurador como GDB es permitir ver qué está pasando "dentro de" otro programa mientras se ejecuta, o lo que el programa estaba haciendo en el momento en que falló.

GDB permite llevar a cabo cuatro tipos de acciones (además de otras cosas en apoyo de éstas) para ayudar a detectar errores en la programación:

- Empezar la ejecución del programa y especificar algo que podrá afectar su conducta.
- Hacer que el programa se detenga en condiciones específicas.
- Examinar lo que pasa, cuando el programa se ha detenido.
- Cambiar diferentes aspectos del programa, para poder experimentar corrigiendo los efectos de un error y seguir para aprender sobre otros.

4.1 Una vista rápida y general de gdb

Para poder depurar un programa con `gdb` es necesario compilarlo con la opción `-g`. Si el programa es `foo.c`, la compilación sería:

```
$ cc foo.c -g -o foo
$
```

Una vez compilado se corre el depurador `gdb` con el programa a depurar como parámetro. Se cuenta con un conjunto de comandos para llevar a cabo dicho análisis. Entre los más importantes están:

help: documentación en línea acerca de los comandos.

file: especifica el programa a depurar

run: empieza a correr el programa bajo `gdb`

break: define un *breakpoint*, es decir, un lugar donde se va a detener la ejecución del programa para verificar los valores de variables o intentar averiguar donde esta fallando el programa

delete: borra todos los *breakpoints* definidos

continue: vuelve a ejecutar el programa de nuevo, después de que se detuvo debido a un breakpoint

step: ejecuta la línea de código actual y detiene la ejecución antes de la siguiente línea de código

next: continua la ejecución hasta antes de la siguiente línea de código; la diferencia con step es que si la siguiente línea es una llamada a una función la función será completamente ejecutada antes de que se detenga la ejecución

until: es parecida a next, excepto que si se encuentra al final de un ciclo, el comando continua la ejecución despueś de que el ciclo se termine, mientras que next empezaría la ejecución al principio del ciclo

list: lista los números de línea del código que se esta depurando

print: imprime el valor de una expresión, que puede ser el nombre de una variable

4.2 Ejemplo de uso de gdb

Para mostrar el uso de algunos de los comandos de `gdb` se usará el siguiente código:

```
int main(void)
{
  int x,y,z;
  float r,s;
```

```
  x=29;
  y=3;
  z = x - y;
  s=10.8;
  r=(3/29) - y;
  printf("Valor de x (%d) y de r (%f) \n",x,r);
}
```

Lo primero es compilar el programa con la opción `-g`. Considerando que el nombre del archivo es `toto.c`, obtendremos:

```
$ cc toto.c -g -o toto
$
```

Como comentario adicional, es bueno saber que la opción `-g` no produce ningún archivo extra.

Después lanzamos `gdb` pasándole como parámetro el nombre del archivo a depurar.

```
$ gdb toto
GNU gdb 4.17
Copyright 1998 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "sparc-sun-solaris2.6"...
(gdb)
```

Si deseamos listar las líneas del programa usamos el comando `list`

```
(gdb) list
1
2     main()
3     {
4       int x,y,z;
5       float r,s;
6
7       x=29;
8       y=23;
9       Z=x-y;
10
(gdb)
```

Si el programa tiene más líneas podemos volver a teclear `list` para ver el resto:

```
(gdb) list
11    s=10.8;
12    r=(3/29) - y;
13
```

```
    14    printf("Valor de x (%d) y de r (%f) \n",x,r);
    15  }
(gdb)
```

El comando `list` nos despliega las diez líneas déspues, o alrededor, de la línea actual.

Ahora bien, si deseamos definir un breakpoint en la línea 7:

```
(gdb) break 7
Breakpoint 1 at 0x80483ce: file toto.c, line 7.
(gdb)
```

Para empezar la ejecución del programa utilizamos el comando `run`. El resultado se presenta abajo:

```
(gdb) run
Starting program: /root/toto

Breakpoint 1, main() at toto.c:7
7       x=29;
(gdb)
```

Para ejecutar la siguiente línea se ejecuta `next`

```
(gdb) next
8       y=3;
(gdb)
```

Si se quiere ir línea por línea se puede ejecutar `n` como abreviación de `next`.

```
(gdb) n
9       z= x - y;
(gdb)
```

Se puede desplegar el tipo y el valor de una variable con los comandos `whatis` y `print`.

```
(gdb) whatis y
type = int
(gdb) print y
$1 = 3
(gdb)
```

También podemos ejecutar la siguiente línea con el comando `step`:

```
(gdb) step
11     s=10.8;
(gdb)
```

Para ejecutar el resto del código sin detenerse se tiene el comando continue `c`:

```
(gdb) c
Continuing
Valor de x (29) y de r (-3.0000000)

Program exited with code 044
(gdb)
```

Por último para salir del depurador se tiene el comando `quit`

```
(gdb) quit
$
```

Como comentario final, al igual que el comando `next` lo pudimos abreviar con una `n`, la mayor parte de los comandos los podemos ejecutar con su primera letra (por ejemplo basta una `s` para ejecutar el comando `setp`).

Dejamos al lector dos preguntas: ¿porqué el programa termina con un código 044? y ¿qué es lo que tendría que hacer para obtener una salida normal?, es decir que el depurador imprima el mensaje:

```
Valor de x (29) y de r (-3.000000)

Program exited normally.
```

4.3 El depurador gdb y los archivos core

Los archivos core se producen cuando ocurre una falla interna dentro de un proceso. Son una imagen del proceso fallido. Contiene toda la información necesaria para una depuración: contenido de los registros de hardware, estatus del proceso y datos del proceso. La utileria `gdb` permite examinar este archivo para determinar donde falló el programa. Haciendo una analogia con el mundo de la aviación, el archivo `core` es la caja negra que se examina cada vez que ocurre un accidente de aviación.

Para poder depurar el archivo `core` con `gdb` es necesario que se cumplan dos condiciones:

1. El programa a analizar debió compilarse con la opción `-g` de depuración.
2. El programa debió abortar y producir un archivo `core`, lo cual generalmente ocurre cuando el programa despliega un mensaje que incluye el enunciado: `(core dumped)`.

Para empezar a analizar el archivo es necesario lanzar `gdb` pasandole como parámetro el archivo ejecutable que falló y el archivo `core`. Si el archivo se llama `toto`, se tendría que teclear:

```
$ gdb toto core
```

La última línea que `gdb` desplegará antes del "prompt" será algo del estilo:

```
#0 0xef607e54 in main() at line 344 in main.cpp
```

lo anterior corresponde al último enunciado que se ejecutó y que probablemente causó la falla.

Sin embargo no siempre se despliega un número de líena. Para averiguar cual función llamó a la función actual, o cual enunciado fue ejecutadoe se puede usar el comando `up`.

El comando `down` realiza lo contrario que el comando `up`. Finalmente, para ver todo el registro de activación se puede usar el comando `backtrace` o su abreviación `bt`.

Para ejemplificar lo anterior usaremos el código siguiente:

```
int main(void)
{
  int a;

  printf("Programa de prueba \n");
  a = 0;
  a++;
  printf("Antes asignacion ");
  a = a/0;
}
```

Compilando y ejecutando:

```
$ gcc calcula.c -g -o calcula
$ calcula
Programa de prueba
Arithmetic Exception (core dumped)
$
```

Para depurar se lanza `gdb` con el nombre del ejecutable y el archivo `core`.

```
$ gdb calcula core
GNU gdb 4.17
Copyright 1998 Free Software Foundation, Inc.
GDB is free software, covered by the GNU General Public License, and you are
welcome to change it and/or distribute copies of it under certain conditions.
Type "show copying" to see the conditions.
There is absolutely no warranty for GDB.  Type "show warranty" for details.
This GDB was configured as "sparc-sun-solaris2.6"...
Core was generated by `calcula'.
Program terminated with signal 8, Arithmetic Exception.
Reading symbols from /usr/lib/libc.so.1...done.
Reading symbols from /usr/lib/libdl.so.1...done.
Reading symbols from /usr/platform/SUNW,Ultra-5_10/lib/libc_psr.so.1...done.
#0  0xff336eac in  ()
(gdb)
```

El último mensaje (`#0  0xff336eac in  ()`) no nos indica la línea en que falló. Para tener más presición del lugar de falla usamos el comando `up`.

```
(gdb) up
```

```
#1  0x106c8 in main () at calcula.c:11
11   a = a/0;
(gdb)
```

Por último si queremos ver todo el stack, usamos `backtrace`

```
(gdb) backtrace
#0  0xff336eac in  ()
#1  0x106c8 in main () at calcula.c:11
(gdb)
```

Para salir de `gdb` usamos `q` la abreviación de `quit`.

```
(gdb) q
$
```

4.4 Conectando gdb a un proceso en ejecución

Es posible conectar un proceso que se esta ejecutando a `gdb`. Al igual que en las secciones anteriores es indispensable que el proceso se haya compilado con la opción de depuración `-g`.

Para atarlo es necesario:

1. Identificar el PID del proceso al que se desea conectar
2. Lanzar `gdb` con el nombre del ejecutable de ese proceso (para tener la información de los símbolos)
3. En `gdb`, ejecutar el comando `attach proceso`, donde `proceso` es el PID del proceso.

Una vez completados estos pasos, `gdb` está en control de la aplicación y puede utilizarse como si se hubiera arrancado desde `gdb`. Una variante es detener el proceso primero (con `Ctrl-Z`), y luego de que `gdb` asuma el control de la aplicación, ejecutar el comando `fg` para reactivarlo de nuevo.

4.5 Otros comandos

Existe una gran cantidad de comandos en `gdb`, los esenciales se presentaron en la sección anterior. Otros comandos que pueden ser útiles son:

cd: para cambiar de directorio

pwd: para identificar el nombre del directorio actual

shell: para salir a un subshell, para regresar a `gdb` basta con teclear `quit` en el subshell

search reg-expr: para buscar en el código fuente una expresion regular a partir de la línea actual hacia abajo.

reverse-search reg-expr: para buscar en el código fuente una expresion regular a partir de la línea actual hacia arriba.

make para correr el programa make.

Aparte de los comandos vistos anteriormente, existe un número mucho más grande de comandos que puden auxiliarnos para depurar un programa.

La mejor forma de conocer todos es a través del comando de ayuda `help` del mismo `gdb`. Este comando no despliega todos los comandos, en su lugar despliega los tipos de comandos con que se cuenta.

```
(gdb) help
List of classes of commands:

aliases -- Aliases of other commands
breakpoints -- Making program stop at certain points
data -- Examining data
files -- Specifying and examining files
internals -- Maintenance commands
obscure -- Obscure features
running -- Running the program
stack -- Examining the stack
status -- Status inquiries
support -- Support facilities
tracepoints -- Tracing of program execution without stopping the program
user-defined -- User-defined commands

Type "help" followed by a class name for a list of commands in that class.
Type "help" followed by command name for full documentation.
Command name abbreviations are allowed if unambiguous.
(gdb)
```

Si se requiere obtener más información acerca de algún tipo de comando, se puede usar el mismo comando `help` seguido del tipo del que se desea obtener más información.

Por ejemplo si deseamos obtener más información acerca de los comandos tipo `running` debemos teclear:

```
(gdb) help running
Running the program.

List of commands:

attach -- Attach to a process or file outside of GDB
continue -- Continue program being debugged
detach -- Detach a process or file previously attached
finish -- Execute until selected stack frame returns
handle -- Specify how to handle a signal
info handle -- What debugger does when program gets various signals
jump -- Continue program being debugged at specified line or address
kill -- Kill execution of program being debugged
next -- Step program
nexti -- Step one instruction
```

```
run -- Start debugged program
set args -- Set argument list to give program being debugged when it is started
set environment -- Set environment variable value to give the program
show args -- Show argument list to give program being debugged when it is started
signal -- Continue program giving it signal specified by the argument
step -- Step program until it reaches a different source line
stepi -- Step one instruction exactly
target -- Connect to a target machine or process
thread -- Use this command to switch between threads
thread apply -- Apply a command to a list of threads
apply all -- Apply a command to all threads
tty -- Set terminal for future runs of program being debugged
unset environment -- Cancel environment variable VAR for the program
until -- Execute until the program reaches a source line greater than the current

Type "help" followed by command name for full documentation.
Command name abbreviations are allowed if unambiguous.
(gdb)
```

Lo cual nos despliega todos los comandos del tipo `running`.

5 El depurador dbx

Esta sección fue extraída del capítulo 13 de [1].

El depurador `dbx` fue desarrollado por Sun y es una utilería propietaria. `gdb` es una utilería de domnio público. Sin embargo tienen la misma filosofía de trabajo y varios comandos en común.

La utilería `dbx` permite monitorear y controlar la ejecución de un programa. Se puede pasar a través de un programa línea por línea mientras se examina el estado de la ejecución del ambiente. También permite examinar archivos `core`. Cuando un error serio ocurre durante la ejecución de un programa, el sistema operativo despliega `Segmentation violation -- core dumped` o un mensaje similar y crea un archivo `core` que contiene información acerca del estado del programa y del sistema cuando ocurrió la falla. Usando `dbx` se puede identificar la línea en el programa donde el error ocurrió, los valores de las variables es este punto. Debido a que los archivos core tienden a ser largos y ocupan mucho espacio en disco, se debe asegurar de removerlos después.

Si se quiere usar el depurador con un programa, se debe usar la opción `-g` cuando se compila un programa. La opción `-g` causa que `cc` genere información adicional que es usada por los depuradores.

Si se trata de depurar un programa que no incluye la tabla simbólica de información que `dbx` espera, se verá un mensaje de advertencia. Se debe recompilar el programa con la apropiada opción y volver a correr `dbx`.

```
$dbx tabs
Reading symbolic information...
Read 12 symbols
warning:  main routine not compiled with the -g option
.
.
$cc -g tabs.c  -o tabs
```

Para correr el depurador en el ejecutable, se debe especificar el nombre del archivo en la línea de comandos cuando se corre `dbx`. Después de que dbx lee el archivo, este guía a la entrada con la cadena `(dbx)`. El comando `list` imprime las primeras diez líneas del código fuente (líneas de comentarios, `#includes`, y `#define` no se incluyen; esos son preprocesados por el preprocesador de C antes de que el programa fuera compilado en forma ejecutable).

```
$ dbx tabs
Reading symbolic information...
Read 63 symbols
(dbx) list
7  int posn = 0;
8 int inc;
9
10 while ((c_getchar()) \@=EOF)
11     switch( c )
12                 {
13 case '\t':
14        inc = findstop (posn);
15        posn += inc;
16        for ( ; inc>0; inc--)
```

Los siguientes comandos checan variables particulares. El comando where reporta cual función está activa, además de su dirección (un número hexadecimal). Su puede usar el comando `whatis` para encontrar la declaración que aplica a una variable en particular (en este caso, la variable col es un entero). Finalmente, se puede usar whereis para localizar una variable; en este ejemplo, la variable `col` fue encontrada en la función findstop en el programa tabs.

```
(dbx) where
main( ) at 0x2290
(dbx) whatis col
int col;
(dbx) whereis col
variable:  'tabs' findstop ' col
```

Una de las más importantes características de un depurador es la habilidad de correr un programa bajo circunstancias controladas. Por ejemplo se puede detener el proceso de corrida en cualquier momento que se quiera checar el estado de algunas variables. La siguiente declaración le pide a `dbx` detener el proceso cada vez que la función `findstop` es llamada. Hecha la requisición, la siguiente declaración corre el programa (tabs, como se especifica en la línea de comando cuando `dbx` empieza) y usa el archivo testtabs como entrada. Cuando el proceso se detiene, se puede usar `print` para checar el valor de una variable en ese punto (`posn`, en este caso) y también usa el comando where para aprender que función está activa y como está siendo llamada. En este ejemplo, el proceso fue detenido en la función `findstop`, como se requirió; `findstop` fue llamado por `main`.

```
(dbx)  stop in findstop
(2) stop in findstop
(dbx) run < testtabs
Running: tabs > testtabs
Stopped in findstop al line 38 in file
"/home/alex/cprogs/tabs.c"
38 return (TABSIZE -  (col  %  TABSIZE));
```

```
(dbx) print posn
posn = 3
(dbx) where
findstop (col = 3), line 38 in "/home/alex/cprogs/tabs.c"
main( ), line 17 in "/home/alex/cprogs/tabs.c"
```

Habiendo examinado las variables de interés, el siguiente comando (`cont`) causa que el proceso siga corriendo. El archivo `testtabs` contiene sólo una línea; el proceso termina de ejecutarse, y los resultados aparecen en la pantalla, siempre y cuando la línea de comando no redireccione la salida. Este depurador reporta que la ejecución se completa y reporta los valores del código de salida. Para terminar la sesión , se usa el comando `quit`.

```
(dbx) cont
xyz abc
execution completed, exit code is 33627
program exited with 91
(dbx) quit
```

La utilería `dbx` soporta muchos comandos que son diseñados para hacer la depuración más fácil. Si no se está seguro de cuales comandos están disponibles, se puede preguntar a `dbx` que los liste usando el comando `help`, como se muestra a continuación. También se puede requerir ayuda más detallada sobre comandos específicos.

```
$dbx
(dbx) help
Command summary
Execution and Tracing
catch clear cont delete ignore next return
run status step stop trace when
.
.
The command 'help <midname> provides additional help for each command
(dbx)
```

6 El depurador ddd

Tan solo unas palabras para el depurador `ddd` antes de dejar esta sección. El nombre `ddd` son las iniciales de data display debugger. Los depuradores `gdb` y `dbx` están basado en texto. Para interactuar con ellos se tienen que introducir los comandos a través del teclado. `ddd` es un depurador interactivo para X-Window. En realidad, ddd utiliza para funcionar un depurador de más bajo nivel, por defecto el gdb de GNU, aunque esto pasa en principio inadvertido al usuario.

`ddd` se encuentra disponible para la mayor parte de las distribuciones de Linux, por lo que para su instalción bastará con usar el comando `rpm` (en RedHat, SuSe, etc) o `dpkg` (en Debian).

Lo más sencillo es invocarlo, sin parámetros, desde la línea de comandos:

```
$ ddd
```

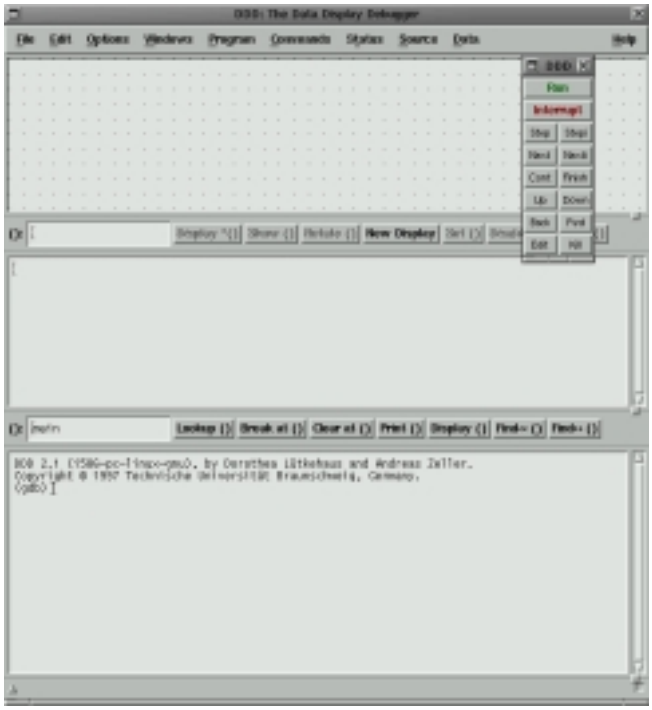

Figura 1: *Las tres ventanas principales de ddd*

Se abrirá una ventana en X-Windows. Pulsando la opción `[File + Open Program...]` podremos abir el programa ejecutable que deseamos depurar. Para ahorrarnos este paso, podemos invocar el depurador de la forma:

```
$ ddd <programa_ejecutable>
```

La ventana de `ddd` está dividida en tres zonas (ver figura 1):

1. *La ventana de datos (data window)*: muestra el contenido de las variables del programa que está siendo depurado. La representación de las variables es cómoda y atractiva visualmente.
2. *La ventana de código fuente (source window)*: muestra (si está disponible) el código fuente del programa depurado.
3. *La consola del depurador (debugger console)*: acepta comandos para el depurador y enseña los mensajes lanzados por éste.

Existen otras ventanas opcionales, como la ventana de herramientas (command tool), con la que se accede a los comandos principales del depurador: `run`, `step`, etc.

Todo lo que se puede hacer con `gdb` se puede hacer con `ddd`. La diferencia esta en la forma de indicarle las acciones y como despliega los resultados. Al igual que `gdb` y `dbx` es necesario que el programa a depurar haya sido compilado con la opción `-g`.

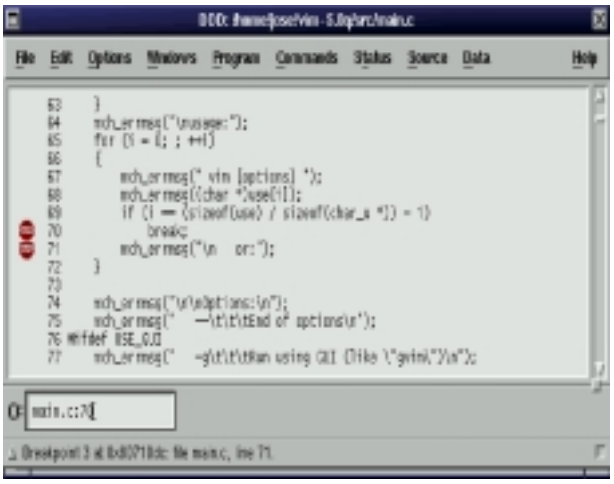

Figura 2: *Ejemplo depuración programa con ddd*

7 La utilería make

Al igual que la sección de `dbx` esta sección fue extraída del capítulo 13 de [1].

Cuando se tiene un programa largo con muchos archivos fuente y de encabezado, los archivos típicamente dependen unos de otros de maneras complejas. Cuando se cambia un archivo del cual otros dependen, se debe recompilar todos los archivos dependientes. Por ejemplo, se pueden tener varios archivos fuente, todos los cuales usan un mismo archivo de encabezado. Cuando se hace un cambio al archivo de encabezado, cada uno de los archivos fuente debe ser recompilado. El archivo de encabezado podría depender de otros archivos, y así en adelante.

Cuando se está trabajando en un programa largo, puede ser difícil, consume tiempo y tedio determinar cuales módulos necesitan ser recompilados debido a sus dependencias. La utilidad make automatiza este proceso.

En su uso más simple, make mira las líneas de dependencia en un archivo llamado makefile en el directorio de trabajo. Las líneas de dependencia indican las relaciones entre los archivos, especificando un archivo objeto que depende en uno o varios archivos prerequisitos. Si se ha modificado algunos de los archivos prerequisitos más recientemente que su archivo objeto, make actualiza el archivo objetio basándose en comandos de construcción que siguen la línea de dependencia. La utilería make normalmente se detiene si encuentra algún error durante el proceso de construcción.

Un makefile sencillo tiene el siguiente formato:

```
target:  prerequisite-list
TAB construction-commands
```

La línea de dependencia está compuesta por el target y la lista de prerequisitos, separado por un guión. La línea de comandos de construcción debe empezar con un caractér de tabulador (al que se designará por `TAB`) y debe seguir la línea de dependencia.

El target es el nombre del archivo que depende de los archivos de la lista de prerequisitos. Los comandos de construcción son regularmente comandos al shell que construye el archivo target. La

utilería make ejecuta los comandos de construcción cuando el tiempo de modificación de uno o varios de los archivos de la lista de prerequisitos es más reciente que el del archivo target (objeto).

El ejemplo de abajo muestra la línea de dependencia y comandos de construcción para el archivo llamado `form`. Este depende de sus prerequisitos, `size.o` y `length.o`. Un comando de construcción apropiado es: de `cc` hace el target.

```
form:        size.o          length.o
             cc -o form size.o  length.o
```

Cada uno de los prerequisitos en una línea de dependencia puede ser un target de otra línea de dependencia. Por ejemplo, ambos, `size.o` y `length.o` son targets en otra línea de dependencia. La red de especificaciones de dependencia puede crear una jerarquía compleja que especifica relaciones entre muchos archivos

El siguiente makefile corresponde a la gráfica completa de dependencia mostrada posteriormente. El archivo ejecutable form depende de dos archivo objeto, y cada archivo objeto depende de sus respectivos archivos fuente y de encabezado, `form.h`. En turno, form.h depende de otros dos archivos de encabezado.

```
$ cat makefile

form:                   size.o          length.o
                        cc -o form size.o  length.o

size.o:                 size.c          form.h
                        cc -c size.c

length.o:               length.c        form.h
                        cc -c length.c

form.h:                 num.h           table.h
                        cat num.h  table.h > form.h
$
```

La última línea ilustra el hecho de que se puede poner cualquier comando de shell Bourne en la línea de construcción. Debido a que los makefiles son procesados por el Shell Bourne, la línea de comando debe ser una que se puede meter como entrada en respuesta al prompt del shell.

7.1 Dependencias implicadas

Se puede confiar en dependencias implicadas y comandos de construcción para hacer el trabajo de escribir el makefile más fácilmente. Si no se incluye una línea de dependencia para un archivo objeto, make asume que depende en el archivo de código fuente del compilador o del ensamblador. Si un prerequisito para un archivo target es xxx.o, y no hay comando de construcción siguiendo a una línea de dependencia para xxx.o target, make busca uno de los siguientes archivos en el directorio de trabajo. Si encuentra un archivo fuente apropiado, make provee de una línea de comando de construcción por default que llama al compilador o ensamblador apropiado para crear el archivo objeto.

Filename	Tipo de Archivo
xxx.c	Código fuente de C
xxx.r	Código fuente de RAFTOR
xxx.f	Código fuente de FORTRAN77
xxx.y	Código fuente de YACC
xxx.l	Código fuente de Lex
xxx.s	Código ensamblador
xxx.mod	Código fuente de Modula 2
xxx.p	Código fuente de Pascal
xxx.sh	Shell scripts

RAFTOR, Modula 2, Pascal y FORTRAN77 son lenguajes de programación estándar, y YACC y Lex son herramientas UNIX para crear lenguajes de comando.

El siguiente ejemplo muestra un makefile que guarda un archivo llamado compute de actualización. Las primeras tres líneas (cada una empieza con `#`) son líneas de comentario. Debido a que `make` ignora líneas que empiecen con `#`, estos signos se pueden usar para comentar. La primera línea de dependencia muestra que compute depende de dos archivos objeto: `compute.o` y `calc.o`. La correspondiente línea de construcción le da al comando make lo que necesita para producir compute. La siguiente línea de dependencia muestra que `compute.o` no sólo depende de su archivo fuente en C sino también de un archivo de encabezado, `compute.h`. La línea de construcción para `compute.o` usa el optimizador de compilación (opción `-O`). El tercer set de líneas de dependencia y construcción no son requeridas. En su ausencia, `make` podría inferir que `calc.o` es dependiente de `calc.c` y produciría la línea de comando necesaria para la compilación.

```
$ cat makefile
#
# makefile for compute
#
compute:                compute.o       calc.o
                        cc -o compute   compute.o    calc.o

compute.o:              compute.c       compute.h
                        cc -c  -O  compute.c

calc.o:                 calc.c
                        cc -c   calc.c

clean:                  rm *.o
$
```

No hay prerequisitos para el último target, `clean`, en el `makefile` de arriba. Este target es comúnmente usado para deshacerse de archivos extraños que pueden estar desactualizados o ser no necesarios, como los archivos `.o`.

Enseguida hay algunos ejemplos de ejecuciones de `make`, basados en el `makefile` anterior. Como el comando `ls` muestra, `compute.o`, `calc.o` y compute no son actualizados. Consecuentemente, el comando `make` corre los comandos de construcción que los actualiza.

```
$ ls -l
total  22
-rw-rw----      1 alex                  179     Jun 21  18:20  calc.c
```

```
-rwxrwx---      1 alex                  354     Jun 21  18:20  calc.o
-rw-rw----      1 alex                  6337    Jun 21  18:20  compute
-rw-rw----      1 alex                  780     Jun 21  18:20  compute.c
-rw-rw----      1 alex                  49      Jun 21  18:20  compute.h
-rw-rw----      1 alex                  880     Jun 21  18:20  compute.o
-rw-rw----      1 alex                  311     Jun 21  18:20  makefile
$ make
    cc -c -O compute.c
    cc -c calc.c
    cc -o compute compute.o calc.o
$
```

Si se corre make otra vez y luego otra vez sin hacer ningún cambio a los archivos prerequisto, make indica que el programa está actualizado sin ejecutar comando alguno.

```
$ make
'compute'  is up to date
$
```

El siguiente ejemplo usa la utilería `touch` para cambiar el tiempo de modificación de un archivo prerequisito. Esta simulación muestra que puede pasar si se hace algún cambio. La utilería make sólo ejecuta los comandos necesarios para hacer la actualización.

```
$ touch calc.c
$ make
    cc -c calc.c
    cc -o compute compute.o calc.o
$
```

En el siguiente ejemplo, `touch` cambia el tiempo de modificación de `compute.h`. La utilería `make` recrea `compute.o` porque depende de `compute.h`, y `make` recrea el ejecutable porque depende de `compute.o`.

```
$ touch compute.h
$ make
    cc -c -O compute.c
    cc -o compute compute.o calc.o
$
```

Como en estos ejemplos, `touch` es útil cuando se quiere usar make para recompilar, o no, programas. Se puede usar para actualizar los tiempos de actualización de todos los archivos fuente así que `make` considera que nada está actualizado. La utilería `make` recompilará todo. Alternativamente, se puede usar touch o la opción `-t` para que make considere todo para actualización. Esto es útil si las modificaciones de tiempo o los archivos han cambiado, así todos los archivos son actualizados.

Una vez que se está satisfecho con el programa que se creó, se puede usar el `makefile` para limpiar los archivos que ya no se necesitan. Es útil guardar archivos intermedios mientras se este escribiendo y depurando el programa, de manera que sólo se necesitan reconstruir los que se necesitan para hacer cambios. Si ya no se va a trabajar en el programa otra vez por un rato, se debe limpiar el disco duro. La ventaja de usar el target clean en el makefile es que no se tiene que recordar todas las pequeñas piezas que se deben borrar. El ejemplo de abajo simplemente borra todos los archivos objeto (`.o`).

```
$ make clean
rm *.o
$
```

Referencias

[1] *A Practical Guide to the Unix Systems*, Sobell, The Benjamin/Cummins Publishing, Company Inc, 1994

[2] *El sitio internet de Gnu:* `http://www.gnu.org`