



**Universidad Nacional
Autónoma de México**
Facultad de Ingeniería



Fundamentos de Programación (1122)

Laboratorios de computación
salas A y B

Profesor: M.I. Marco Antonio Martínez Quintana
Semestre 2021-1

Practica No. 10

Depuración de Programas

Grupo: 1129

No. de Equipo de cómputo empleado: No aplica

No. de Lista o Brigada: No aplica

No. de Lista: 42

Nombre: Adolfo Román Jiménez

Objetivo:

Aprender las técnicas básicas de depuración de programas en C para revisar de manera precisa el flujo de ejecución de un programa y el valor de las variables; en su caso, corregir posibles errores.

Introducción:

La depuración de programas es el proceso de identificar y corregir errores de programación. En inglés se conoce como debugging, porque se asemeja a la eliminación de bichos (bugs), manera en que se conoce informalmente a los errores de programación.

Si bien existen técnicas para la revisión sistemática del código fuente y se cuenta con medios computacionales para la detección de errores (depuradores) y facilidades integradas en los sistemas lower CASE y en los ambientes de desarrollo integrado, sigue siendo en buena medida una actividad manual, que desafió a la paciencia, la imaginación y la intuición de programadores. Muchas veces se requiere incluir en el código fuente instrucciones auxiliares que permitan el seguimiento de la ejecución del programa, presentando los valores de variables y direcciones de memoria y ralentizando la salida de datos ("modo de depuración"). Dentro de un proceso formal de aseguramiento de la calidad, puede ser asimilado al concepto de "prueba unitaria"

La depuración de un programa es útil cuando:

- Se desea optimizar el programa
- El programa tiene algún fallo
- El programa tiene un error de ejecución o defecto

Algunas funciones básicas que tienen en común la mayoría de los depuradores son las siguientes:

- Ejecutar el programa: se procede a ejecutar el programa en la herramienta de depuración ofreciendo diversas opciones para ello.
- Mostrar el código fuente del programa: muestra cuál fue el código fuente del programa con el número de línea con el fin de emular la ejecución del programa sobre éste, es decir, se indica qué parte del código fuente se está ejecutando a la hora de correr el programa.
- Punto de ruptura: también conocido por su traducción al inglés breakpoint, sirve para detener la ejecución del programa en algún punto indicado previamente por medio del número de línea. Como la ejecución del programa es más rápida de lo que podemos visualizar y entender, se suelen poner puntos de ruptura para conocer ciertos parámetros de la ejecución como el valor de las variables en

determinados puntos del programa. También sirve para verificar hasta qué punto el programa se ejecuta sin problemas y en qué parte podría existir el error, esto es especialmente útil cuando existe un error de ejecución.

- Continuar: continúa con la ejecución del programa después del punto de ruptura.
- Ejecutar la siguiente instrucción: cuando la ejecución del programa se ha detenido por medio del depurador, esta función permite ejecutar una instrucción más y detener el programa de nuevo. Esto es útil cuando se desea estudiar detalladamente una pequeña sección del programa. Si en la ejecución existe una llamada a función se ingresará a ella.
- Ejecutar la siguiente línea: es muy similar a la función anterior, pero realizará todas las instrucciones necesarias hasta llegar a la siguiente línea de código. Si en la ejecución existe una llamada a función se ignorará.
- Ejecutar la instrucción o línea anterior: deshace el efecto provocado por alguna de las funciones anteriores para volver a repetir una sección del programa.
- Visualizar el valor de las variables: permite conocer el valor de alguna o varias variables.

Dependiendo de la herramienta usada para compilar el programa, si es de consola o de terminal, su uso y las funciones disponibles variarán.

Depuración de programas escritos en C con GCC y GDB

Para depurar un programa usando las herramientas desarrolladas por GNU, éste debe compilarse con información para depuración por medio del compilador GCC.

Para compilar, por ejemplo, un programa llamado *calculadora.c* con GCC con información de depuración, debe realizarse en una terminal con el siguiente comando:

```
gcc -g -o calculadora calculadora.c
```

El parámetro *-g* es quien indica que el ejecutable debe producirse con información de depuración.

Una vez hecho el paso anterior, debe usarse la herramienta GDB, la cual, es el depurador para cualquier programa ejecutable realizado por GCC.

Para depurar un ejecutable debe invocarse a GDB en la terminal indicando cuál es el programa ejecutable a depurar, por ejemplo, para depurar *calculadora*:

```
gdb ./calculadora
```

Al correr GDB se entra a una línea de comandos. De acuerdo al comando es posible realizar distintas funciones de depuración:

```
~/ $ gcc -g -o calculadora calc.c
~/ $ gdb ./calculadora
Reading symbols from ./calculadora...
(gdb) █
```

list o *l*: Permite listar diez líneas del código fuente del programa, si se desea visualizar todo el código fuente debe invocarse varias veces este comando para mostrar de diez en diez líneas. Se puede optar por colocar un número separado por un espacio para indicar a partir de qué línea desea mostrarse el programa. También es posible mostrar un rango de líneas introduciendo el comando y de qué línea a qué línea separadas por una coma. Ejemplo: *list 4,6*

```
~/ $ gcc -g -o calculadora calc.c
~/ $ gdb ./calculadora
Reading symbols from ./calculadora...
(gdb) list 4,6
4
5     int main ()
6     {
(gdb) list
7         int op, uno, dos;
8
9         do
10        {
11            printf(" --- Calculadora ---\n");
12            printf("\n¿Qué desea hacer\n");
13            printf("1) Sumar\n");
14            printf("2) Restar\n");
15            printf("3) Multiplicar\n");
16            printf("4) Dividir\n");
(gdb) █
```

b: Establece un punto de ruptura para lo cual debe indicarse en qué línea se desea establecer o bien también acepta el nombre de la función donde se desea realizar dicho paso. Ejemplo: *b main*

```

~/ $ gdb ./calculadora
Reading symbols from ./calculadora...
(gdb) b main
Breakpoint 1 at 0x11a9: file calc.c, line 6.
(gdb) b 10
Breakpoint 2 at 0x11c4: file calc.c, line 11.
(gdb) b 40
Breakpoint 3 at 0x134a: file calc.c, line 40.
(gdb) 

```

do delete: Elimina un punto de ruptura, indicando cuál es el que debe eliminarse usando el número de línea.

```

~/ $ gdb ./calculadora
Reading symbols from ./calculadora...
(gdb) b main
Breakpoint 1 at 0x11a9: file calc.c, line 6.
(gdb) b 43
Breakpoint 2 at 0x1370: file calc.c, line 44.
(gdb) d 44
No breakpoint number 44.
(gdb) d 43
No breakpoint number 43.
(gdb) 

```

clear: Elimina todos los puntos de ruptura. Ejemplo: *clear*

```

~/ $ gdb ./calculadora
Reading symbols from ./calculadora...
(gdb) b main
Breakpoint 1 at 0x11a9: file calc.c, line 6.
(gdb) b 43
Breakpoint 2 at 0x1370: file calc.c, line 44.
(gdb) d 44
No breakpoint number 44.
(gdb) d 43
No breakpoint number 43.
(gdb) clear
No source file specified.
(gdb) 

```

info line: Permite mostrar información relativa a la línea que se indique después del comando. Ejemplo: *info line 43*

```
~/ $ gdb ./calculadora
Reading symbols from ./calculadora...
(gdb) info line 43
Line 43 of "calc.c" is at address 0x1370 <main+455> but contains no code.
(gdb) info line 44
Line 44 of "calc.c" starts at address 0x1370 <main+455> and ends at 0x137c <main+467>.
(gdb) □
```

run o *r*: Ejecuta el programa en cuestión. Si el programa tiene un punto de ruptura se ejecutará hasta dicho punto, de lo contrario se ejecutará todo el programa.

```
(gdb) r
Starting program: /home/ubuntu/calculadora
--- Calculadora ---

¿Qué desea hacer
1) Sumar
2) Restar
3) Multiplicar
4) Dividir
5) Salir
□
```

c: Continúa con la ejecución del programa después de un punto de ruptura.

```

Reading symbols from ./calculadora...
(gdb) b main
Breakpoint 1 at 0x11a9: file calc.c, line 6.
(gdb) b 44
Breakpoint 2 at 0x1370: file calc.c, line 44.
(gdb) r
Starting program: /home/ubuntu/calculadora

Breakpoint 1, main () at calc.c:6
6      {
(gdb) c
Continuing.
--- Calculadora ---

¿Qué desea hacer
1) Sumar
2) Restar
3) Multiplicar
4) Dividir
5) Salir

```

s: Continúa con la siguiente instrucción después de un punto de ruptura.

```

Breakpoint 1, main () at calc.c:6
6      {
(gdb) s
11      printf(" --- Calculadora ---\n");
(gdb) s
__GI_IO_puts (str=0x555555556008 " --- Calculadora ---") at ioputs.c:33
33      ioputs.c: No such file or directory.
(gdb) s
35      in ioputs.c
(gdb) s

```

n: Salta hasta la siguiente línea de código después de un punto de ruptura.

```

~/ $ gdb ./calculadora
Reading symbols from ./calculadora...
(gdb) b main
Breakpoint 1 at 0x11a9: file calc.c, line 6.
(gdb) b 34
Breakpoint 2 at 0x1311: file calc.c, line 34.
(gdb) r
Starting program: /home/ubuntu/calculadora

Breakpoint 1, main () at calc.c:6
6      {
(gdb) n
11      printf(" --- Calculadora ---\n");
(gdb) 

```

p o *print*: Muestra el valor de una variable, para ello debe escribirse el comando y el nombre de la variable separados por un espacio. Ejemplo: *p suma_acumulada*

```

(gdb) n
Introduzca los números a restar separados por comas
2      scanf("%d, %d",&uno, &dos);
(gdb) n
3      printf("%d - %d = %d\n", uno, dos, (uno - dos));
(gdb) p uno
1 = 5
(gdb) p dos
2 = 6
(gdb) 

```

ignore: Ignora un determinado punto de ruptura indicándolo con el número de línea de código. Ejemplo: *ignore 5*

```

~/ $ gdb ./calculadora
Reading symbols from ./calculadora...
(gdb) b main
Breakpoint 1 at 0x11a9: file calc.c, line 6.
(gdb) b 5
Note: breakpoint 1 also set at pc 0x11a9.
Breakpoint 2 at 0x11a9: file calc.c, line 6.
(gdb) r
Starting program: /home/ubuntu/calculadora

Breakpoint 1, main () at calc.c:6
6      {
(gdb) ignore 5
Second argument (specified ignore-count) is missing.
(gdb) ignore main
bad breakpoint number: 'main'
(gdb) 

```

q o *quit*: Termina la ejecución de GDB.


```
(gdb) q
A debugging session is active.

    Inferior 1 [process 4732] will be killed.

Quit anyway? (y or n) y
~/ $
```

GDB tiene más opciones disponibles que pueden consultarse con comandos como *help* o invocando desde la terminal del sistema *man gdb*.

```
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 all" for the list of all commands.
Type "help" followed by command name for full documentation.
Type "apropos word" to search for commands related to "word".
Type "apropos -v word" for full documentation of commands related to "word".
--Type <RET> for more, q to quit, c to continue without paging--
```

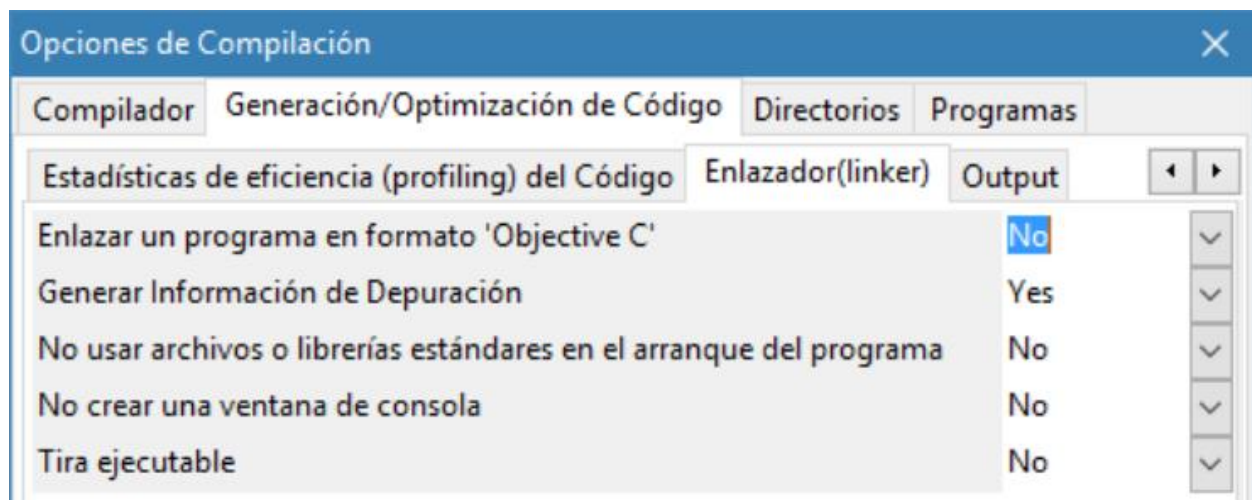
Depurador de programas escritos en C con Dev-C++ 5.0.3.4

Dev-C++, es una IDE especializada para desarrollar programas escritos en C o C++. Si bien incorpora un editor de textos y un compilador integrados, también posee un depurador. Cabe destacar que por defecto Dev-C++ se basa en el compilador GCC y el depurador en GDB, aunque de manera gráfica ello es transparente para el usuario ya que todo simula una sola herramienta.

Cabe señalar, que si se desea utilizar Dev-C++ como herramienta de desarrollo de programas en C, debe estar instalado adecuadamente en el equipo para que funcione tanto el compilador como las herramientas de depuración. Si se usa sistema operativo Windows, se recomienda encarecidamente usar la versión que se proporciona en <http://lcp02.fib.unam.mx> en la sección de *Servicios*.

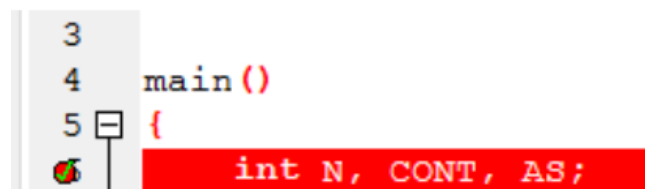
Antes de iniciar la depuración debe tenerse a la mano el archivo con el programa escrito en C o proceder a escribirlo en la misma IDE. Para ello debe usarse el menú *Archivo* → *Nuevo* → *Código Fuente* si se piensa usar la IDE para escribirlo o en su lugar *Archivo* → *Abrir Proyecto* o *Archivo* si ya existía el código fuente.

Una vez que se tiene el programa, debe activarse la opción de compilación generando información para el depurador. Para activar esta opción debe abrirse el menú *Herramientas* → *Opciones del Compilador* y acceder a la pestaña *Generación/Optimización de Código* y finalmente, en la subpestaña Enlazador (linker), activar la opción *Generar Información de Depuración*:



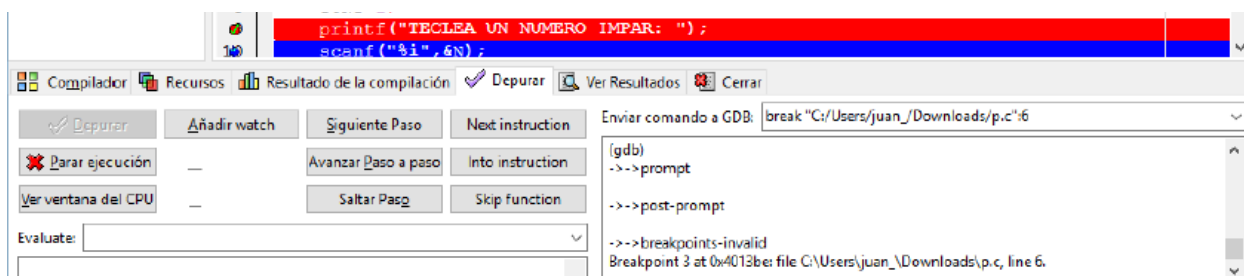
Después de realizar lo anterior, el programa realizado puede compilarse y ejecutarse con lo que ofrece el menú *Ejecutar*.

Para agregar puntos de ruptura, debe hacerse clic en la línea de código donde se desea colocar y ésta se volverá en color rojo. Para retirarlo se hace clic de nuevo en la línea y volverá a su color normal. Lo anterior se ve en la imagen siguiente:



Para depurar el programa, primero se debe compilar con el menú *Ejecutar* → *Compilar* y luego depurar con *Depurar* → *Depurar*. El programa se abrirá y se ejecutará hasta el primer punto de ruptura seleccionado. También se abrirá un cuadro de herramientas en la parte inferior del programa que tiene las principales herramientas de depuración en la

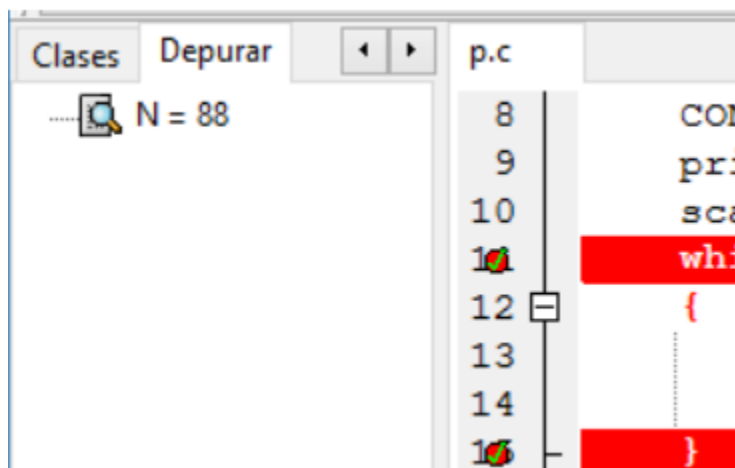
parte derecha. Cabe destacar que la línea que se ejecuta actualmente es la que se resalta en color azul:



Cuando se llega al punto de ruptura, se tienen diversas opciones, *Siguiete Paso* ejecuta la siguiente línea (si existe una iteración o función, la saltará), *Avanzar Paso a Paso* ejecuta instrucción por instrucción (una función o iteración serán ejecutadas instrucción por instrucción), *Saltar Paso* ejecuta hasta el siguiente punto de ruptura.

Para detener la depuración puede seleccionarse la opción *Parar ejecución*. La opción *Ver ventana del CPU* permite ver a detalle las instrucciones enviadas al procesador, registros de memoria involucrados y valor de cada una de las banderas en el procesador.

Finamente, para estudiar el valor de cada variable, se puede recurrir a la función *Añadir Watch* y escribir el nombre de la variable. En un cuadro a la izquierda, se verá el nombre de la variable y su valor hasta el punto donde se está ejecutando el programa:



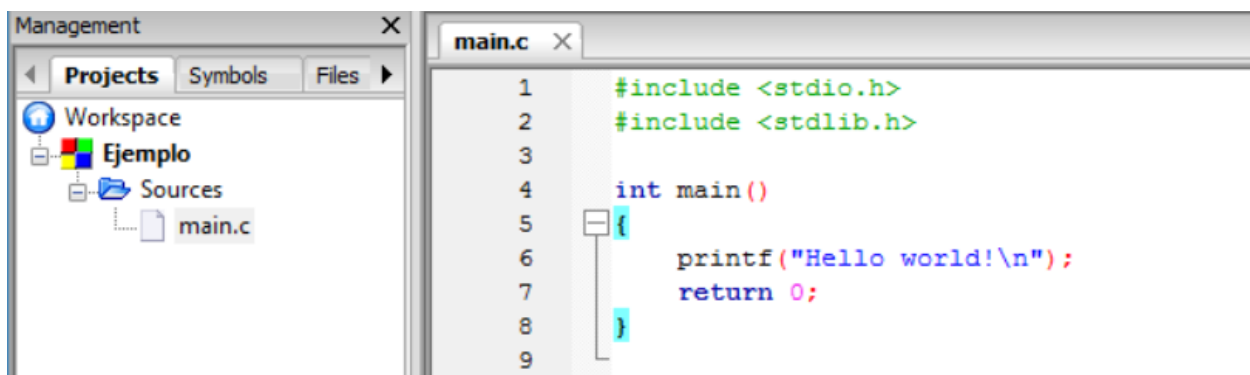
Cuando se utiliza esta IDE, se recomienda usar los accesos directos mencionados en los propios menús, lo cual permite usarla de manera óptima. El nombre de las funciones y menús pueden variar según el idioma en el que se haya instalado la IDE.

Depuración de programas escritos en C con Code::Blocks 13.12

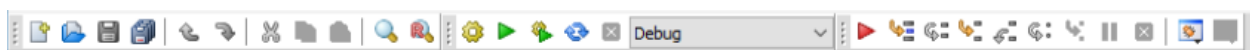
Code::Blocks, es otra IDE de código abierto que puede basarse en las mismas herramientas GNU que Dev-C++. Permite cambiarse por otros motores de compilación si se desea. Proporciona un editor de texto, un compilador integrado, herramientas de depuración, etc.

Para poder depurar, es necesario crear un nuevo proyecto desde el menú *File* → *New* → *Project* y elegir en el cuadro que aparece *Console Application* (recordar que, por ahora, todos los programas son desarrollados en modo de consola). Seguir el asistente para crear el proyecto eligiendo que se usará el lenguaje C, luego seleccionar un título adecuado para el proyecto, la ruta donde se creará y el título del archivo asociado al proyecto. Posteriormente, en el mismo asistente seleccionar *Create “Debug” Configuration* y *Create “Release” Configuration* en el mismo asistente con compilador *GNU GCC Compiler*. Nótese que existen dos carpetas asociadas que son */bin/debug* y */bin/release* que son donde se crearán los ejecutables de depuración y el final respectivamente. Se debe finalizar el asistente.

En la parte izquierda, se encontrará el nombre del archivo fuente del proyecto. Para ello navegar como se indica en la siguiente imagen y dar clic en *main.c*:



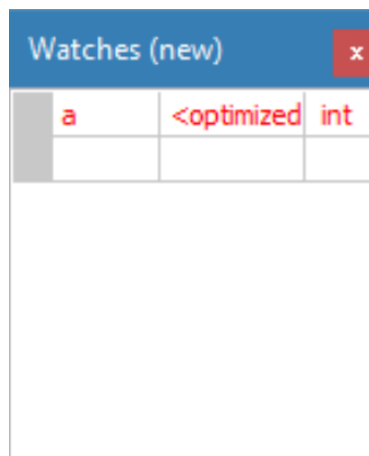
Dicho archivo debe editarse para formar el programa deseado. Cuando se está desarrollando es importante que esté seleccionado el modo de depuración, localizado en la barra de herramientas que se muestra:



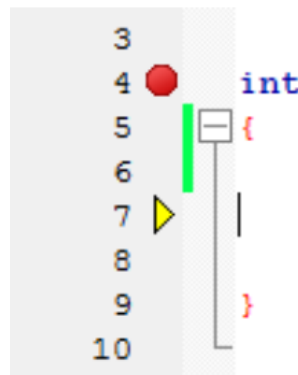
Como puede observarse, hay un menú despegable que tiene la opción *Debug* y *Release*, debe estar siempre seleccionada la primera opción hasta no haber terminado el programa a desarrollar. Cuando todo esté listo, se cambiará a la segunda opción y se usará el archivo ejecutable que se localiza en la */bin/release*, jamás el localizado en */bin/debug* que solo tiene efectos de desarrollo.

A la izquierda del menú, se encuentran las opciones de compilación y ejecución del programa en el modo seleccionado. A la derecha se encuentran las opciones de depuración. La primera opción, *Debug/Continue*, permite ejecutar el programa en modo de depuración y reanudar la ejecución después de un punto de ruptura. La opción *Stop debugger*, detiene la depuración y permite continuar editando. Existen otras herramientas adicionales, entre ellas correr el programa hasta donde se encuentre el cursor en el texto, ejecutar la siguiente línea o la siguiente instrucción, todas ellas estudiadas anteriormente.

Para visualizar una variable, debe hacerse clic sobre ella con el depurador corriendo, y dar clic en *Watch 'variable'*, aparecerá un pequeño cuadro con la tabla de las variables que se desean visualizar y su valor:



Finalmente, para agregar un punto de ruptura, se tiene que hacer clic derecho sobre el número de línea de código y agregarlo, aparecerá un punto rojo en la línea, para quitarlo se tiene que hacer el mismo procedimiento.



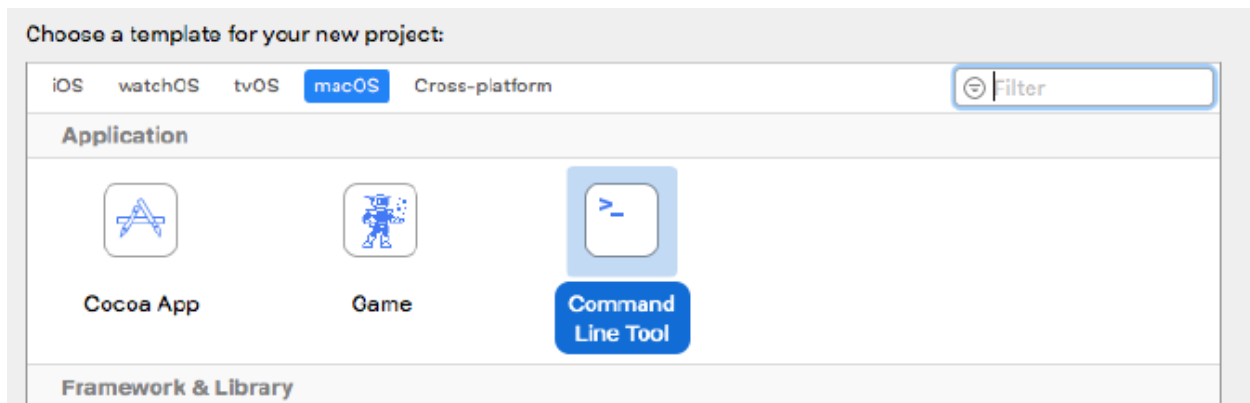
La parte del código que se está ejecutando por el depurador, se indica por medio de una flecha amarilla en el número de línea correspondiente.

Depuración de programas escritos en C con Xcode de Mac

Xcode es otro Entorno de Desarrollo Interactivo (IDE) que, al igual que los antes mencionados en esta guía, además de contener herramientas de depuración, cuenta con

un editor y un compilador entre otros. Es por esto que para aplicar las actividades de depuración de un programa en C en equipos MAC utilizando esta herramienta, debemos realizar lo siguiente.

Abrir la aplicación Xcode, crear un nuevo proyecto seleccionando **Create a new Xcode project**; a lo cual aparecerá la siguiente pantalla donde se deberá seleccionar **macOS** y **Command Line Tool**, y dar **Next**.



Posteriormente dar las características del proyecto, seguido de **Next**. Por ejemplo:

Product Name:

Team:

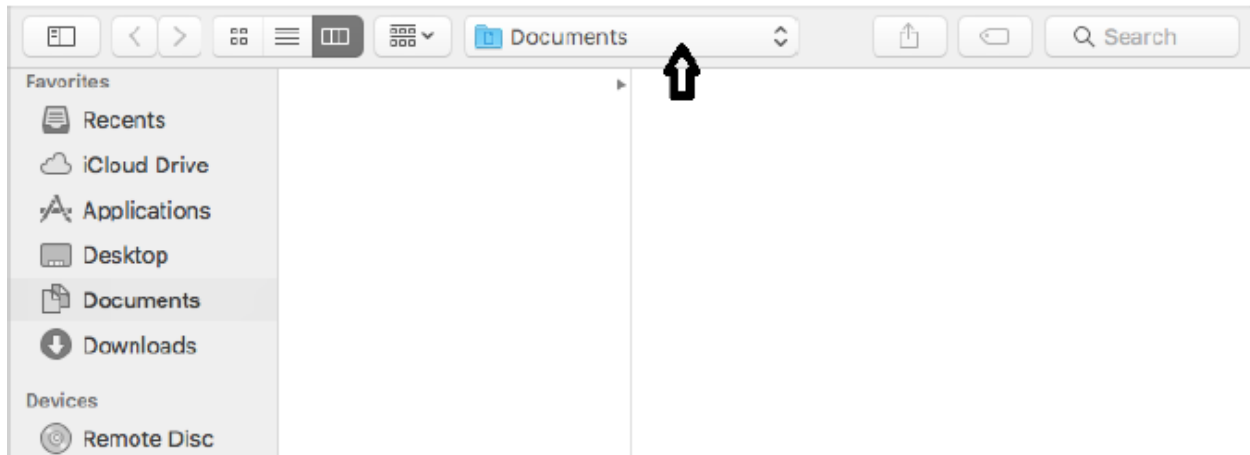
Organization Name:

Organization Identifier:

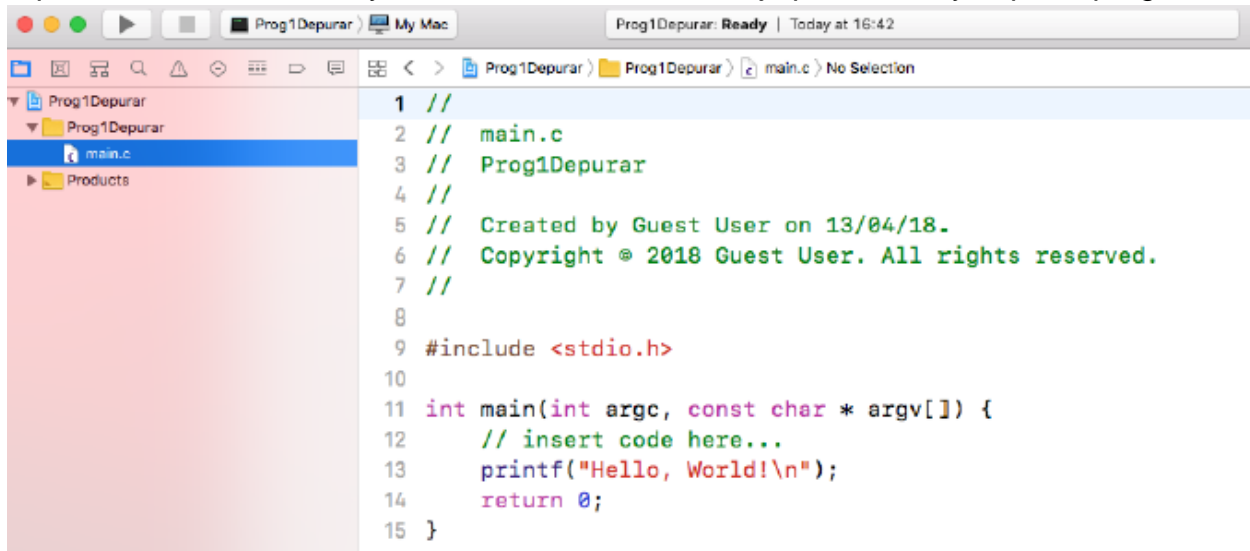
Bundle Identifier: unam.Prog1Depurar

Language:

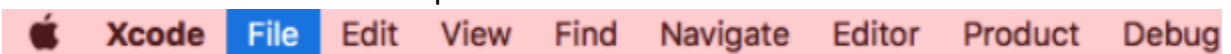
Se presentará una ventana donde hay que indicar dónde se grabará el proyecto. Por ejemplo en Documents. Después dar **Create**.



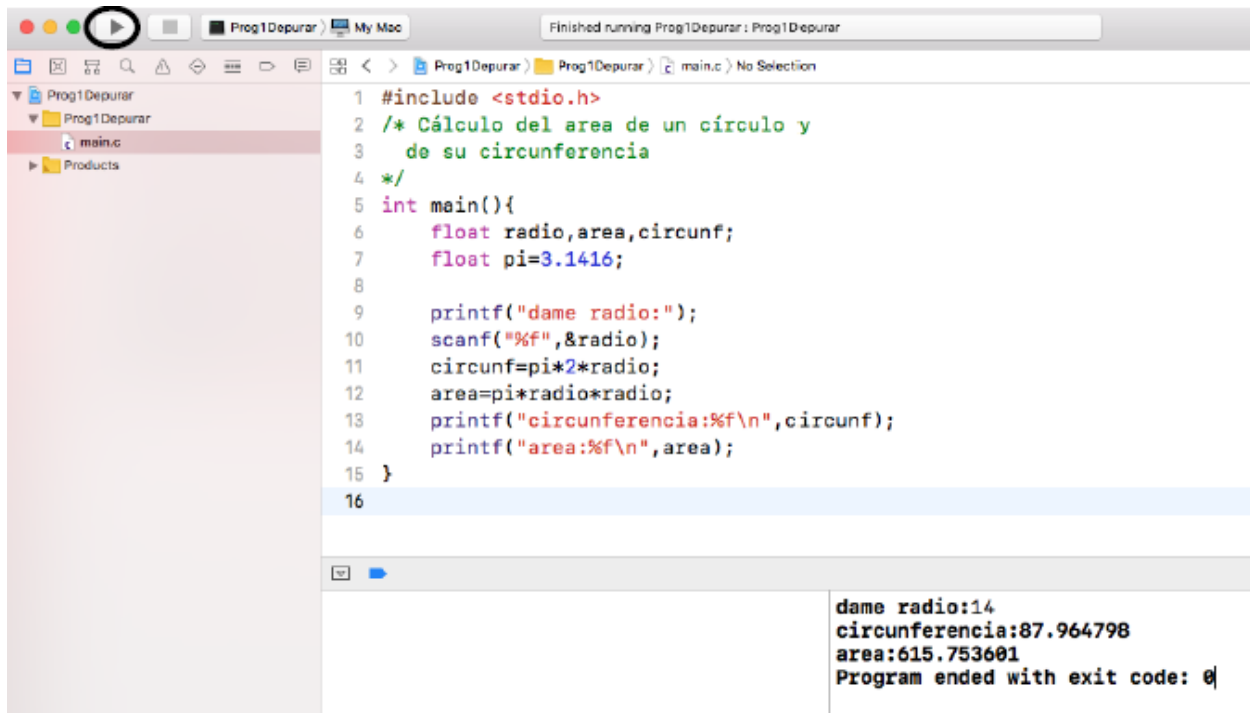
A partir de este momento ya tenemos el área de trabajo para editar y depurar programas:



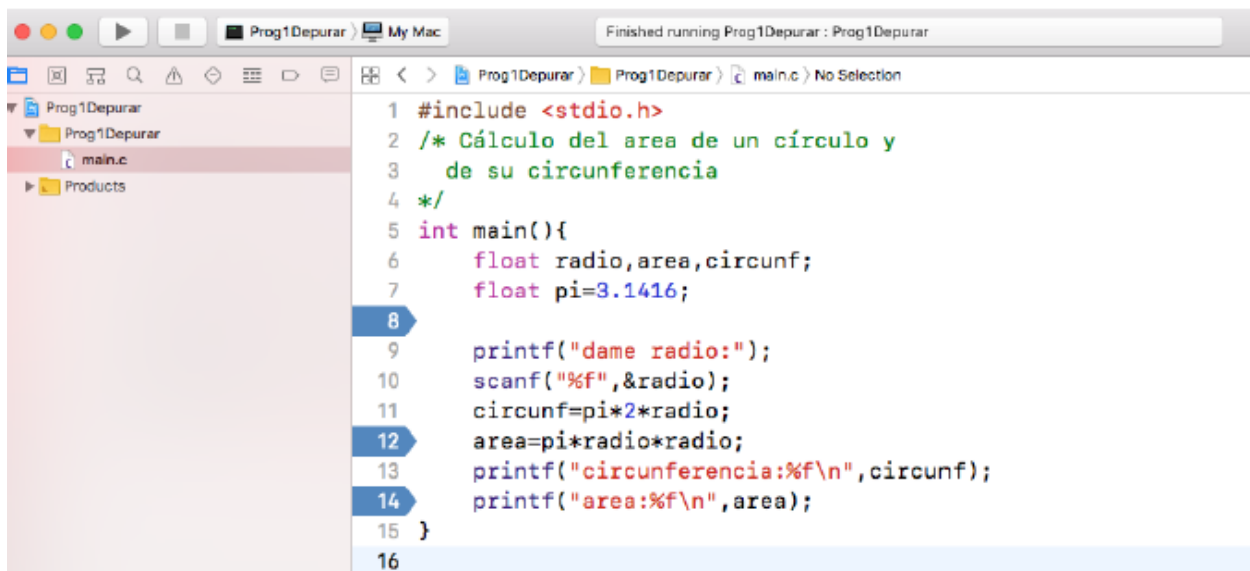
Seleccionando main.c en la jerarquía de archivos, mostrado en el lado izquierdo, podremos observar que nos presenta un “esqueleto” de un programa en C, el cual se deberá sustituir por el que se vaya a depurar. Como Xcode es un IDE, podemos aquí mismo editar y compilar antes de iniciar las actividades de depuración. Para ello nos podemos auxiliar de la barra superior de menús de Xcode:



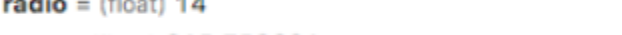
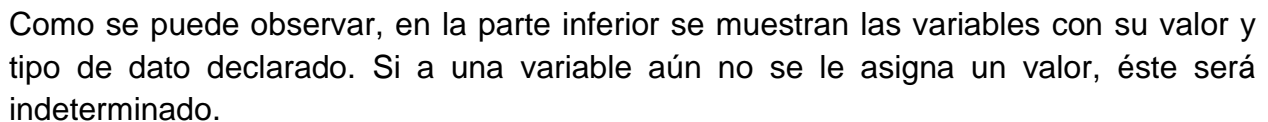
Una vez que se tiene el programa a depurar editado, primeramente, se recomienda compilarlo antes de realizar su depuración; esto se realiza usando las teclas Cmd +B. Posteriormente para ejecutarlo (o compilarlo y ejecutarlo si no se ha hecho anteriormente la compilación), en la barra de trabajo utilizar ►



Para iniciar con algunas actividades de depuración, vamos a indicar puntos de ruptura. Esto se realiza haciendo clic, con el botón derecho del mouse, sobre el número que indica la línea de código donde se desea detener y aparecerá una flecha azul:



Una vez definidos los puntos de ruptura, al momento de ejecutar el programa se detendrá al encontrar el primer punto de ruptura, indicando automáticamente el valor de todas las variables definidas en el programa hasta ese punto. Por ejemplo:



The screenshot shows a Python IDE with a toolbar at the top containing icons for file operations, execution, and debugging. Below the toolbar, the following code is visible in the editor:

```
radio = (float) 14
area = (float) 615.753601
circunf = (float) 87.9647979
pi = (float) 3.14159989
```

- Se recomienda revisar las opciones que se tienen en el menú Debug para realizar otras actividades de depuración.

Debug	Source Control	Window	Help
Pause			⌘Y
Continue To Current Line			⌘C
Step Over			F6
Step Into			F7
Step Out			F8
Step Over Instruction			⌘F6
Step Over Thread			⌘⇧F6
Step Into Instruction			⌘F7
Step Into Thread			⌘⇧F7
Capture GPU Frame			
Capture GPU Scope			▶
GPU Overrides			▶
Simulate Location			▶
Simulate Background Fetch			
Simulate UI Snapshot			
iCloud			▶
View Debugging			▶
Activate Breakpoints			⌘Y
Breakpoints			▶
Debug Workflow			▶
Attach to Process by PID or Name...			
Attach to Process			▶
Detach from Prog1Depurar			

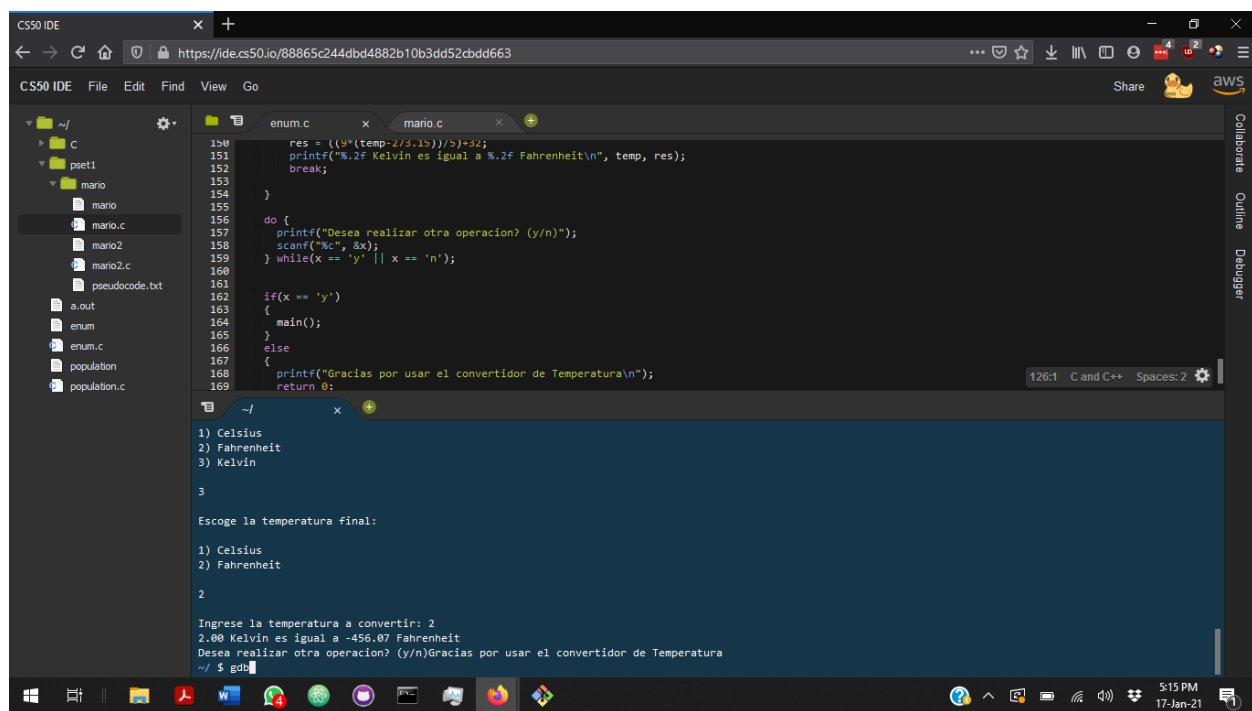
Actividades:

- Revisar, a través de un depurador, los valores que va tomando una variable en un programa escrito en C, al momento de ejecutarse.
- Utilizando un depurador, revisar el flujo de instrucciones que se están ejecutando en un programa en C, cuando el flujo depende de los datos de entrada.

En esta práctica use la antepenúltima versión de la practica 8 debido a que tiene un problema que no pude resolver, así que pensé que sería bueno intentar usar el GDB para ver si puedo entender que es lo que mi programa está haciendo mal.

El problema principalmente es que el programa después de finalizar el segundo switch, sorpresivamente pasa por alto las instrucciones siguientes y termina su ejecución.

1.- Podemos ver que el programa termina su ejecución.



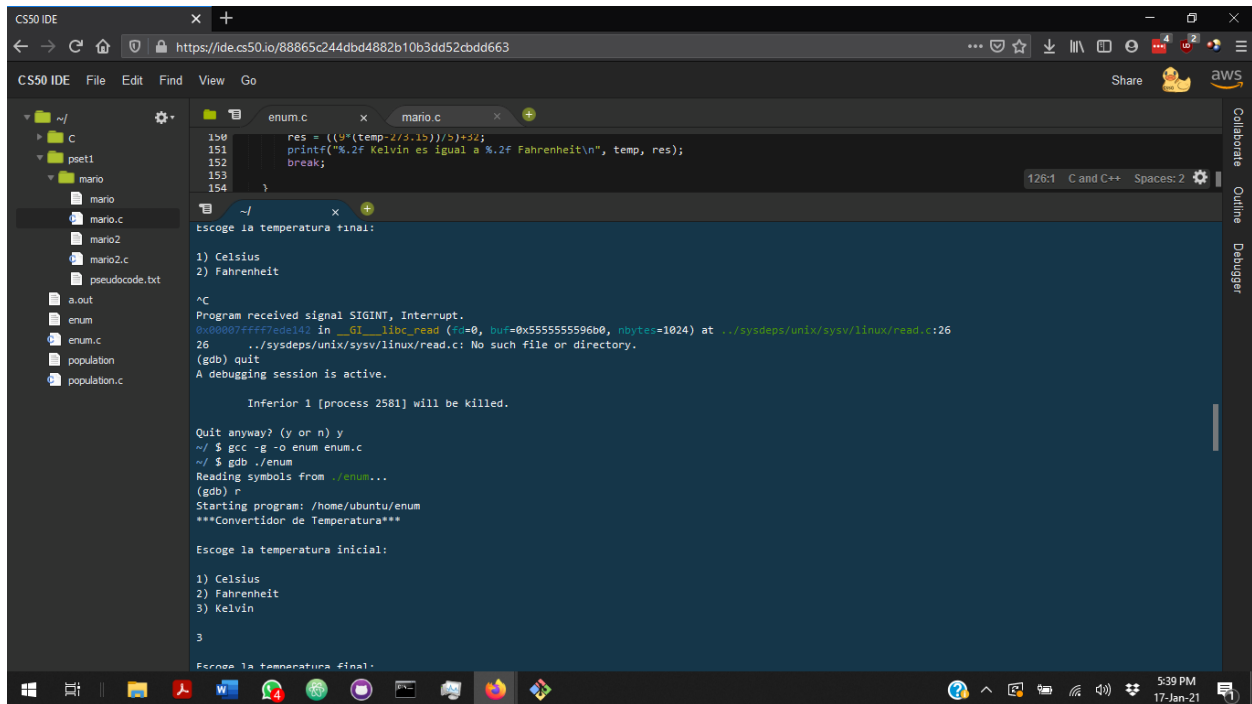
The screenshot shows the CS50 IDE interface. The top panel displays the source code for a C program named 'mario.c'. The code includes a switch statement for temperature conversion and a loop for repeated use. The bottom panel shows the program's execution output, which matches the expected behavior: it prompts for a temperature, converts it, and asks if the user wants to perform another operation.

```
enum.c x mario.c x
150
151     res = ((9*(temp-273.15))/5)+32;
152     printf("%.2f Kelvin es igual a %.2f Fahrenheit\n", temp, res);
153     break;
154 }
155
156 do {
157     printf("Desea realizar otra operacion? (y/n)");
158     scanf("%c", &x);
159 } while(x == 'y' || x == 'n');
160
161 if(x == 'y')
162 {
163     main();
164 }
165 else
166 {
167     printf("Gracias por usar el convertidor de Temperatura\n");
168     return 0;
169 }
```

Output:

```
1) Celsius
2) Fahrenheit
3) Kelvin
3
Escoge la temperatura final:
1) Celsius
2) Fahrenheit
2
Ingrese la temperatura a convertir: 2
2.00 Kelvin es igual a -456.07 Fahrenheit
Desea realizar otra operacion? (y/n)Gracias por usar el convertidor de Temperatura
~/ $ gdb
```

2.- En la siguiente imagen usamos `gcc -g -o enum enum.c` para compilar el programa y que este tenga símbolos de depuración.



```
enum.c
150 res = ((5*(temp-273.15))/9)+32;
151 printf("%.2f Kelvin es igual a %.2f Fahrenheit\n", temp, res);
152 break;
153 }
154 }

~/
Escoge la temperatura inicial:
1) Celsius
2) Fahrenheit
^C
Program received signal SIGINT, Interrupt.
0x0000ffff7edd142 in __GI___libc_read (fd=0, buf=0x555555596b0, nbytes=1024) at ../sysdeps/unix/sysv/linux/read.c:26
26 ../sysdeps/unix/sysv/linux/read.c: No such file or directory.
(gdb) quit
A debugging session is active.

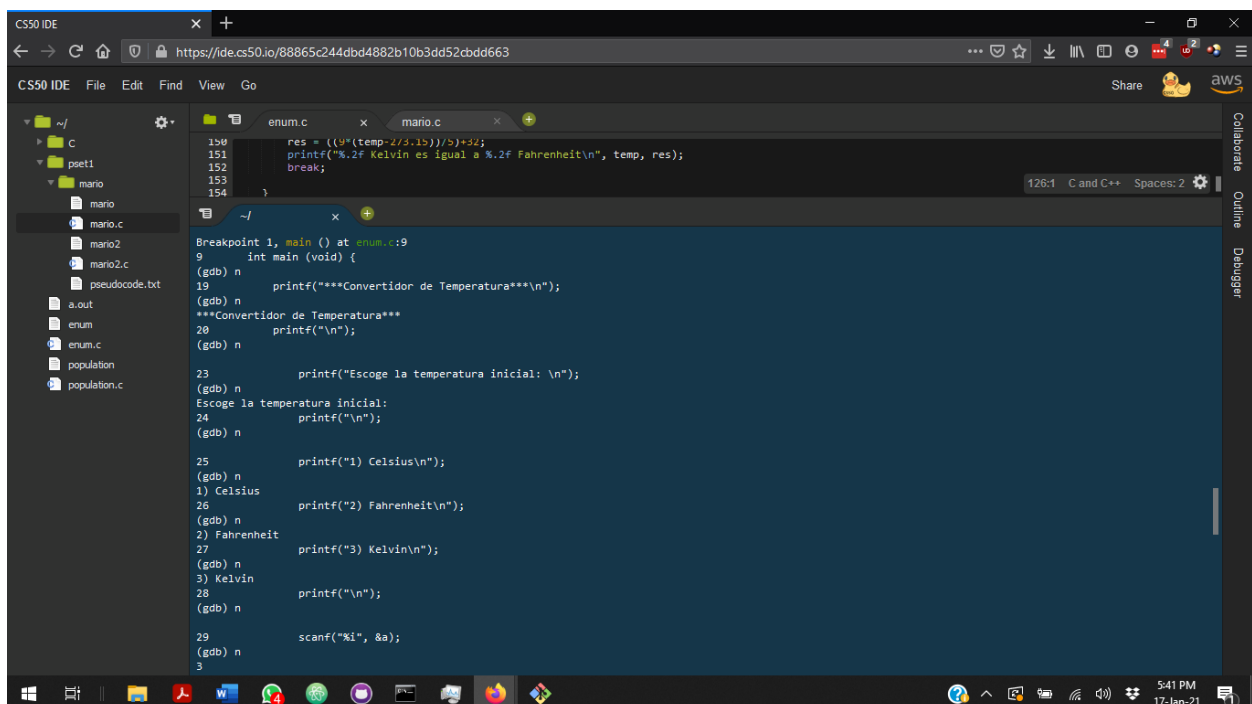
Inferior 1 [process 2581] will be killed.

Quit anyway? (y or n) y
~/ $ gcc -g -o enum enum.c
~/ $ gdb ./enum
Reading symbols from ./enum...
(gdb) r
Starting program: /home/ubuntu/enum
***Convertidor de Temperatura***

Escoge la temperatura inicial:
1) Celsius
2) Fahrenheit
3) Kelvin
3

Escoge la temperatura final:
```

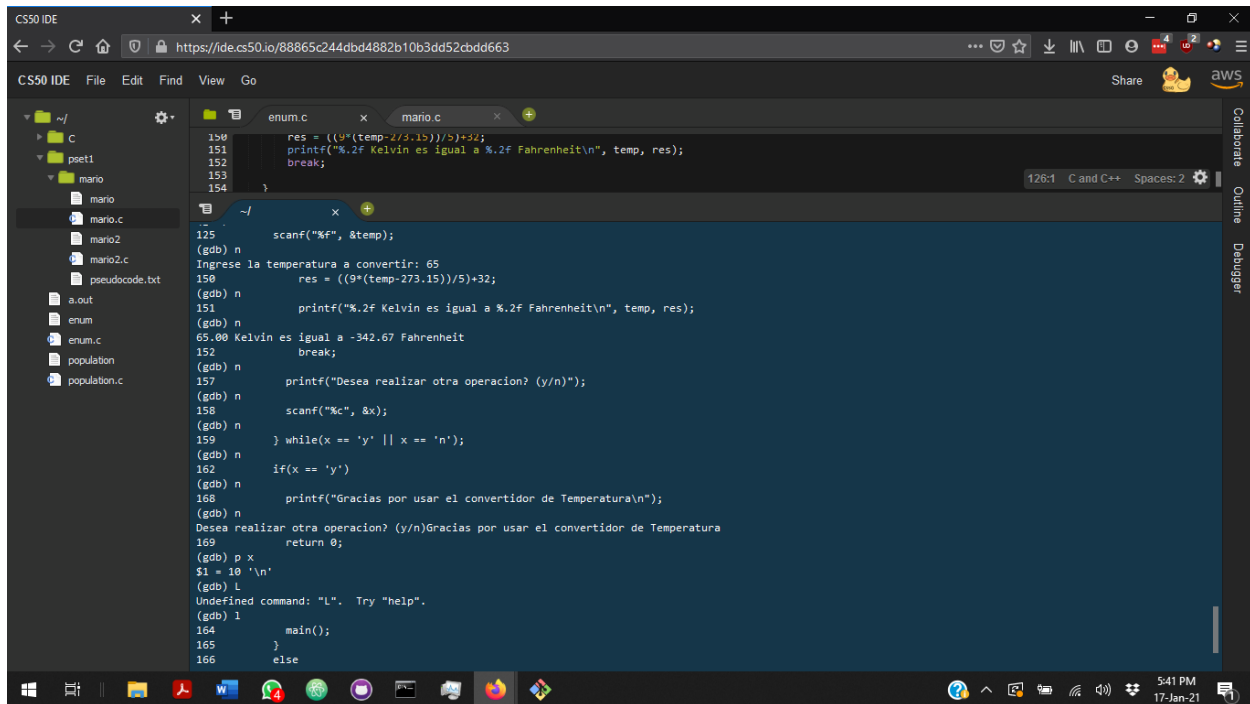
3.- Ejecutamos el programa hasta el punto donde buscamos el error



```
enum.c
150 res = ((5*(temp-273.15))/9)+32;
151 printf("%.2f Kelvin es igual a %.2f Fahrenheit\n", temp, res);
152 break;
153 }
154 }

~/
Breakpoint 1, main () at enum.c:9
9 int main (void) {
(gdb) n
printf("***Convertidor de Temperatura***\n");
(gdb) n
***Convertidor de Temperatura***
20 printf("\n");
(gdb) n
23 printf("Escoge la temperatura inicial: \n");
(gdb) n
Escoge la temperatura inicial:
24 printf("\n");
(gdb) n
25 printf("1) Celsius\n");
(gdb) n
1) Celsius
26 printf("2) Fahrenheit\n");
(gdb) n
2) Fahrenheit
27 printf("3) Kelvin\n");
(gdb) n
3) Kelvin
28 printf("\n");
(gdb) n
29 scanf("%i", &a);
(gdb) n
3
```

4.- Al momento de usar la función p para que imprima el valor de x encontramos que esta tiene un valor: 10 '\n'



The screenshot shows the CS50 IDE interface. The left sidebar displays a file explorer with a project named 'mario' containing files like 'mario.c', 'mario2.c', 'pseudocode.txt', 'a.out', 'enum.c', and 'population.c'. The main editor window shows the code for 'mario.c'. The code includes a function to convert Fahrenheit to Kelvin and a loop to ask the user if they want to perform another operation. The debugger is active, showing the execution flow. The current line of execution is line 164, which is the start of the 'main()' function. The variable 'x' is shown with a value of 10, which is the ASCII value of the newline character '\n'. The debugger output shows the program's execution, including the input of 65 for the temperature conversion and the subsequent loop iteration.

```
enum.c x mario.c
154 res = ((9*(temp-273.15))/5)+32;
155 printf("%.2f Kelvin es igual a %.2f Fahrenheit\n", temp, res);
156 break;
157 }
158
159 scanf("%f", &temp);
160 (gdb) n
161 Ingrese la temperatura a convertir: 65
162 res = ((9*(temp-273.15))/5)+32;
163 (gdb) n
164 printf("%.2f Kelvin es igual a %.2f Fahrenheit\n", temp, res);
165 65.00 Kelvin es igual a -342.67 Fahrenheit
166 break;
167 (gdb) n
168 printf("Desea realizar otra operacion? (y/n)");
169 (gdb) n
170 scanf("%c", &x);
171 (gdb) n
172 } while(x == 'y' || x == '\n');
173 (gdb) n
174 if(x == 'y')
175 (gdb) n
176 printf("Gracias por usar el convertidor de Temperatura\n");
177 (gdb) n
178 Desea realizar otra operacion? (y/n)Gracias por usar el convertidor de Temperatura
179 return 0;
180 (gdb) p x
181 $1 = 10 '\n'
182 (gdb) l
183 Undefined command: "L". Try "help".
184 (gdb) l
185 164     main();
186     }
187     else
```

5.- Mientras investigamos porque x tiene un valor de 10 '\n' encontramos que e valor en ASCII que tiene '\n' es de 10, es por esta razón que la función scanf al momento de leer el valor de x, lo que encuentra es el valor de una nueva línea, lo que provoca que se aplique un "enter" y el programa corra hasta el final consecuentemente.

6.- Procedemos a la corrección del programa.

7.- Después de hacer las correcciones pertinentes programa podemos ver que scanf se detiene al momento de solicitar una entrada y actúa de acuerdo al dato ingresado.

```
148     break;
149     case Kf: //Convierte K -> F
150         res = ((9*(temp-273.15))/5)+32;
151         printf("%.2f Kelvin es igual a %.2f Fahrenheit\n", temp, res);
152         break;
153     }
154
155     do {
156         printf("Desea realizar otra operacion? (y/n)");
157         scanf(" %c", &x);
158     } while(x != 'y' && x != 'n');
159
160     if(x == 'y')
161     {
162         main();
163     }
164     else
165     {
166         printf("Gracias por usar el convertidor de Temperatura\n");
167         return 0;
168     }
169 }
170
171
172 }
```

65.00 Kelvin es igual a -342.67 Fahrenheit
Desea realizar otra operacion? (y/n)

8.- Terminamos con la depuracion.

Conclusión:

Para esta practica aprendi algo bastante interesante sobre como poder hacer un debugging propiamente dicho en C y en la terminal. Fue muy buena esta practica porque inicialmente no tenia idea de que es lo que tenia mi programa o porque estaba funcionando de forma incorrecta y tampoco tenia una forma real de saberlo, solo podía medio imaginarme en donde estaba el problema, pero a partir de la practica, fue que pude entender porque mi programa no estaba funcionando de la manera adecuada y como es que funciona en realidad la función scanf y cuales son las soluciones posibles a ese problema.

Me gustó mucho esta práctica.

Referencias:

- Facultad de ingeniería - UNAM. Manual de prácticas del laboratorio de Fundamentos de programación: http://odin.fib.unam.mx/salac/practicasFP/MADO-17_FP.pdf
- Introduction to GDB a tutorial - Harvard CS50: <https://www.youtube.com/watch?v=sCtY--xRUyl&t=27s>
- OnlineGDB: online compiler and debugger for c/c++: <https://www.onlinegdb.com/>