Práctica 12
Depurador de Turbo C

Objetivo
Aprender el uso básico del depurador de Turbo C por medio de un ejemplo sencillo que muestra el uso de la principales funciones del depurador y aplicando lo aprendido a la depuración de un programa de prueba.

Introducción
Los depuradores son programas que nos permiten ejecutar de manera controlada otro programa. Normalmente se utilizan para eliminar errores que se manifiestan en un comportamiento incorrecto del programa en depuración al tiempo de ejecución. Podemos decir que hay dos categorías de depuradores:

- Depuradores a nivel ensamblador o lenguaje máquina.
- Depuradores a nivel de código fuente.

Los depuradores a nivel de código fuente operan a nivel del lenguaje de alto nivel, que como sabemos tienen una correspondencia 1 a muchos con el lenguaje máquina. Para ser realmente útiles, los depuradores a nivel lenguaje fuente deben incluir las siguientes funciones: *Puntos de ruptura* (Breakpoints), *Ejecución normal*, *Avanzar una instrucción a la vez (step), Ir a la siguiente sentencia (Next)*, entre otras.

Una de las características de los IDEs es la inclusión de un depurador. El IDE del Turbo C no es la excepción. Incorpora un depurador de código fuente. Para ilustrar el uso de este depurador, vamos a usar un ejemplo sencillo de un programa que imprime una lista de numeros y sus cuadrados.

1.- Ilustración del uso del depurador de Turbo C
1.1.- Captura el siguiente programa:

```
/* Archivo: cuad.c
 * Despliega una lista de numeros y sus cuadrados
 */

#include<stdio.h>
/* Prototipos */
void cuadrado(int);

int main(void)
{
      int i;
      for(i=0;i<10;i++)
      {
            printf("i = %d\t",i);
            cuadrado(i);
      }
      return 0;
}

void cuadrado(int n)
{
      int j;
      j = n*n;
      printf("i*i = %d\n",j);
}
```

1.2.- Compílalo con las teclas **ALT+F9** ¿Compilado exitosamente? Sí: ____ No: ____

Debió compilarse exitosamente, si no fue así, lista los mensajes de error mostrados en la ventana de Mensajes y corrígelos:

_____
_____
_____
_____

1.3.- Ejecución paso a paso (trace into)
Esta opción ejecuta una sentencia del programa cada vez. Esta opción se invoca tecleando **F7** (ó la opción Run → Trace into)

Pulsa **F7** varias veces y observa como el resalte pasa de una línea a otra. Observa que cuando se llama a la función `cuadrado(i)`, el resalte pasa a las instrucciones de la función.

Para observar mejor los efectos de la ejecución del programa vamos a activar la ventana de salida con la opción Window → Output y redimensionarla para poder ver simultáneamente la ventana de edición y la ventana de salida (las ventanas del IDE se redimensionan "arrastrando" alguna de sus esquinas).

Para detener la depuración del programa debemos teclear **CTRL+F2** (ó Run → Program reset).

1.4.- Ejecución de la siguiente sentencia (step over)
Esta opción ejecuta una sentencia de la función `main()` cada vez. Esta opción se invoca tecleando **F8** (ó la opción Run → Step over). Cada vez que se invoque esta función se ejecutará otra sentencia, pero sin trazar las llamadas a función.

Pulsa **F8** varias veces y observa como el resalte pasa de una línea a otra. Observa que cuando se llama a la función `cuadrado(i)`, en esta opción el resalte no pasa a las instrucciones de la función.

1.5.- Puntos de ruptura
Aunque la ejecución paso a paso es muy útil, en muchas ocasiones es más sencillo y menos tedioso establecer un punto de ruptura al inicio de alguna sección sospechosa. Cuando la ejecución alcanza el punto de ruptura, el programa deja de ejecutarse y el control vuelve al depurador, permitiendo comprobar el valor de ciertas variables o empezar a ejecutar el programa paso a paso.

Para establecer un punto de ruptura, hay que llevar el cursor a la línea adecuada del programa y activar el punto de ruptura con **CTRL+F8** (ó la opción Debug → Toggle breakpoint). La línea de código donde se ha establecido un punto de ruptura se muestra en fondo rojo, en alta intensidad o en otro color, dependiendo del tipo de video. Se pueden tener activos varios puntos de ruptura en el programa. Una vez definido uno o más puntos de ruptura, el programa se ejecuta usando las teclas **CTRL+F9** (ó la opción Run → Run).

Para eliminar un punto de ruptura, hay que suspender la depuración (tecleando **CTRL +F2**), colocar el cursor en el punto de ruptura que se quiere eliminar y teclear **CTRL+F8** (ó la opción Debug → Toggle breakpoint).

Vamos a establecer un punto de ruptura en la instrucción

```
printf("i*i = %d\n",j);
```

dentro de la funcion `cuadrado(int n)`. Observa como una vez establecido el punto de ruptura, la instrucción cambia de color. Ahora vamos a ejecutar el programa usando las teclas **CTRL+F9** (ó la opción Run→ Run) varias veces y observar como la ejecución siempre se detiene en la instrucción marcada como punto de ruptura.

1.6.- Visualización de variables

Mientras se depura, normalmente se requiere visalizar el valor que toma una o más variables a medida que evoluciona la ejecución del programa. Esto es muy fácil de hacer usando el depurador. Para establecer una variable a visualizar, pulsa **CTRL+F7** (ó la opción Debug → Watches → Add watch), en la ventana que surge, introduce el nombre de la variable que se quiere ver. Si se trata de una variable global, su valor estará siempre disponible, sin embargo, cuando se trata de una variable local, sólo aparecerá su valor cuando se esté ejecutando la función que contiene esa variable.

Como ejemplo, vamos a activar la visualización de la variable `j`. Pulsa **CTRL+F7** y en la ventana que surge teclea **j**. Si no se está ejecutando el programa o la ejecución se ha detenido fuera de la función `cuadrado(int n)`, se verá el mensaje:

```
j: Undefined symbol 'j'
```

Sin embargo, cuando esté en ejecución la función `cuadrado(int n)`, se verá el valor de **j**.

1.7.- Visualización de la pila

Durante la ejecución de un programa se puede ver el contenido de la pila de llamadas usando las teclas **CTRL+F3** (ó la opción Debug → Call stack). Esta opción muestra el orden en el que se llama a las diversas funciones del programa. También muestra el valor de los parámetros de la función en el momento de la llamada. Hay que tener en cuenta que en la pila de llamadas sólo se muestran las funciones programadas. Las llamadas a funciones de biblioteca no se registran.

Para ilustrar el uso de esta función, ejecuta el programa paso a paso y teclea **CTRL+F3** (ó la opción Debug → Call stack) en cada instrucción.

2.- Depuración del programa Insert Sort

Para aplicar lo visto en el ejercicio enterior, vamos a usar un programa de ordenamiento que utiliza el Insert Sort, cuyo objetivo es ordenar ascendentemente los `NumInputs` elementos de un arreglo `Y[]`.

Pseudocódigo Insert Sort:

```
Inicializa arreglo Y[] vacio
Obtener NumInputs números de la línea de comandos
Para I = 1 to NumInputs
  Obtener nuevo elemento NewY
  Encontrar primer Y[J] para el cual NewY < Y[J]
  Recorrer Y[J],Y[J+1],... hacia la derecha, para hacer espacio a NewY
  Y[J] = NewY
FinPara
```

El programa que implementa este algoritmo utiliza el siguiente árbol de llamadas:

`main()`→`GetArgs()`→`ProcessData()`→`Insert()`→`ScootOver()`→`PrintResults()`.

El programa `inserror.c` implementa este pseudocódigo, pero contiene errores lógicos que deben depurarse apoyándose en el depurador del IDE de Turbo C.

2.1.- Captura el siguiente código con el editor del IDE de Turbo C:

```
/* Archivo: inserror.c
  programa que implementa el insert sort con varios errores  */
// variables globales.
int X[10],        // arreglo de entrada
    Y[10],        // Arreglo para trabajo
    NumInputs,    // longitud arreglo de entrada
    NumY = 0;     // numero actual de elementos en Y[].

void GetArgs(int AC, char **AV)
{  int I;
   NumInputs = AC - 1;
   for (I = 0; I < NumInputs; I++)
      X[I] = atoi(AV[I+1]);
}
void ScootOver(int JJ)
{  int K;
   for (K = NumY-1; K > JJ; K++)
      Y[K] = Y[K-1];
}
void Insert(int NewY)
{  int J;
   if (NumY = 0)  {   // Sí Y[] vacio, caso facil
      Y[0] = NewY;
      return;
   }
   // necesario insertar justo antes del primer elemento de Y[]
   // que sea mayor o igual que NewY
   for (J = 0; J < NumY; J++)  {
      if (NewY < Y[J])  {
         // recorre Y[J], Y[J+1],... hacia la derecha antes de insertar NewY
         ScootOver(J);
         Y[J] = NewY;
         return;
      }
   }
}
void ProcessData()
{
   for (NumY = 0; NumY < NumInputs; NumY++)
      // inserta newY en el lugar adecuado entre Y[0],...,Y[NumY-1]
      Insert(X[NumY]);
}
void PrintResults()
{  int I;
   for (I = 0; I < NumInputs; I++)
      printf("%d\n",Y[I]);
}

int main(int Argc, char ** Argv)
{
   GetArgs(Argc,Argv);
   ProcessData();
   PrintResults();
}
```

2.2.- Compilación del programa `inserror.c`

2.1.- Una vez capturado del programa, compílalo con la opción **`Alt+F9`** y describe brevemente los errores y/o advertencias (warnings) que reporte el IDE:

Linea: _____
Error ó Advertencia: _____
_____

Linea: _____
Error ó Advertencia: _____
_____

Linea: _____
Error ó Advertencia: _____
_____

Linea: _____
Error ó Advertencia: _____
_____

Linea: _____
Error ó Advertencia: _____
_____

Linea: _____
Error ó Advertencia: _____
_____

Linea: _____
Error ó Advertencia: _____
_____

2.2.- Corrigiendo el código apoyándonos en los mensajes de error

Utilizando el ratón o la tecla **F6**, si el ratón no responde, colócate en la Ventana de Mensajes y con las teclas de Flechas, recorre los errores o advertencias reportados. Observa que conforme te colocas en los mensajes, en la ventana de edición el resalte se mueve a la instrucción reportada con error.

Varias de las primeras advertencias se deben a la falta de archivos de cabecera y falta de definición de prototipos.

2.3.- Agrega las siguientes líneas al inicio del archivo para eliminar estos errores o advertencias:

```
#include<stdio.h>
#include<stdlib.h>
void GetArgs(int,char**);
void ScootOver(int);
void Insert(int);
void ProcessData(void);
void PrintResults(void);
```

Salva el archivo ya actualizado pulsando **`F2`** y vuelve a compilarlo con **`Alt+F9`**. Después de esta

segunda compilación ya deben quedar cuando mucho 2 errores o advertencias, descríbelos brevemente y corrígelos:

Linea: _____
Error ó Advertencia: _____
Corrección realizada: _____

Linea: _____
Error ó Advertencia: _____
Corrección realizada: _____

2.3.- Una vez corregidos todos los errores de sintaxis, vamos a generar el ejecutable tecleando **F9** (o la opción Compile →Make) y a probar la ejecución del programa desde una ventana de DOS. Ya generado el ejecutable `inserror.exe`, y desde una nueva ventana de DOS, primero nos colocamos en el subdirectorio `c:\tc\bin` usando el comando:

`cd \tc\bin<Enter>`

Ahora ejecutamos el programa con los siguientes valores 17 4 5 200 3 4 17 20 tecleando:

`inserror 17 4 5 200 3 4 17 20<Enter>`

Si el programa se ejecuta adecuadamente, debe desplegar la lista de números ordenados ascendentemente. ¿El programa se ejecuta correctamente? Sí: _____ No: _____

2.4.- Buscando los errores.
Como ya se comentó, el programa contiene errores lógicos, para los cuales el compilador no genera ningún mensaje que pueda ayudarnos. Se genera un ejecutable que al momento de correrse no produce los resultados esperados. Vamos a depurar el programa para localizar y corrregir sus errores. Ya que este programa está escrito para recibir sus datos desde la línea de comandos, para depurarlo desde el IDE de Turbo C debemos pasar estos datos desde dentro del IDE. Para esto usamos la opción Run →Arguments y en la ventana que aparece tecleamos los datos de prueba: **`17 4 5 200 3 4 17 20`** y así podemos iniciar la depuración con estos datos. Para comenzar, coloca un punto de ruptura en la instrucción:

```
ProcessData();
```

dentro de `main()` que representa más o menos la mitad del programa, teclea **`Ctrl + F9`** (o la opción Run →Run). Observa como la ejecución se detiene justo antes de ejecutar la instrucción `ProcessData();`. A partir de este punto, ejecuta paso a paso con **`F7`** (ó la opción Run → Trace into) el código de la función y observa los valores de las variables `Y`, `NumY` y `NumInputs`, traza también la ejecución de la función `Insert(X[NumY])` y observa la pila de llamadas para verificar sus argumentos. Si todo parece estár bien, entonces movemos el punto de ruptura hacia adelante a la instrucción `PrintResults();`, si sospechamos que algo no está bien, entonces movemos el punto de ruptura hacia atrás a la instrucción `GetArgs(Argc,Argv);`.

En cada caso, es conveniente ejecutar hasta el punto de ruptura y trazar paso a paso a partir de ahí observando las variables más pertinentes en cada caso y observando la pila de llamadas a funciones donde aplica.

Trata de localizar todos lo errores que contiene el programa para lograr que se ejecute

correctamente. Recuerden que una vez localizado un punto de error, es necesario diagnosticar la causa, corregirla, compilar el programa, probarlo y, de ser necesario, volver al proceso de depuración.

2.5.- Listado de errores encontrados y correcciones realizadas

Linea: _____
Error ó Advertencia: _____
Corrección realizada: _____

Linea: _____
Error ó Advertencia: _____
Corrección realizada: _____

Linea: _____
Error ó Advertencia: _____
Corrección realizada: _____

Linea: _____
Error ó Advertencia: _____
Corrección realizada: _____

Linea: _____
Error ó Advertencia: _____
Corrección realizada: _____

Linea: _____
Error ó Advertencia: _____
Corrección realizada: _____

3.- Comentarios y conclusiones

_____
_____
_____
_____
_____
_____
_____
_____
_____
_____
_____