Universidad Nacional del Litoral
Facultad de Ingeniería y Ciencias Hídricas
Departamento de Informática

FUNDAMENTOS DE PROGRAMACIÓN

Asignatura correspondiente al plan de estudios de la carrera de Ingeniería Informática

Material Adicional
Herramientas de Depuración

Pablo Novara - 28/05/2019

Herramientas de Depuración

"Si [tu programa] funciona correctamente al primer intento, corre a contarle a tus amigos y vayan a celebrar – porque es algo que no pasa todos los años."

Bjarne Stroustrup

Introducción

Depuración es el proceso de corregir los errores encontrados en un programa. Los errores de un programa (o "bug", como se los denomina comúnmente en la jerga) pueden clasificarse en tres categorías:

- "**De sintaxis**" o "**en tiempo de compilación**": Son aquellos que resultan de un código mal escrito según las normas del lenguaje C++. En general son trivialmente fáciles de detectar porque impiden que se complete el proceso de compilación[1], y por ende el propio compilador identifica el punto exacto en el código fuente en donde se manifiesta el problema; aunque no siempre el mensaje de error de compilador torna evidente la verdadera causa.
- "**En tiempo de ejecución**": Son aquellos que impiden que el programa finalice con normalidad; los que hacen que se detenga abruptamente. Podemos notarlo por un mensaje del sistema operativo, o porque el código de salida no es cero (el 0 del "return 0;" del final de la función "main").
- "**De lógica**": Aún cuando el programa sea sintacticamente válido, y su ejecución llegue hasta el final sin errores, arrojando resultados, estos resultados podrían no ser correctos. Por ejemplo, si se ha utilizado una fórmula matemáticamente válida pero incorrecta/inapropiada para un determinado cálculo.

Nos centraremos en este documento en los errores del segundo y tercer tipo, los que están asociados a la ejecución del programa[2]. Existen herramientas que nos permiten observar detalladamente cómo evolucionan los datos a medida que avanza la ejecución de un programa, y analizar qué acciones o instrucciones del código fuente son las que en verdad se ejecutan en cada paso. Es decir, con estas herramientas podemos pausar la ejecución en un punto dado, avanzar paso a paso observando por cuales lineas del código va pasando el control del programa, averiguar cuánto valen las variables o expresiones en cada momento, etc. La mayoría de los IDEs actuales incluyen herramientas y facilidades

1 Entiéndase aquí "compilación" como el proceso completo de generación del ejecutable. En la práctica, se debe poder distinguir claramente la etapa de compilación propiamente dicha de la etapa de enlazado; ya que los errores generados serán diferentes como así también sus posibles causas.

2 Suponiendo el escenario en que no se tiene otra pista anterior respecto a la causa del problema. En particular, si la compilación arrojó warnings (advertencias), se recomienda primero analizar/resolver dichos warnings (tratarlos como si fueran en realidad errores), ya que muchos errores de lógica (como por ej, utilizar una variable sin inicializar) suelen ser diagnosticados primero por este mecanismo.

específicas para la depuración. En esta guía se utilizará el IDE *Zinjal* para los ejemplos presentados, pero los conceptos generales son aplicables en cualquier otro entorno.

Es importante destacar que más allá de la idea general que dicta que la depuración sirve para encontrar y corregir errores; en muchos casos la depuración será también de utilidad aún cuando el programa funcione correctamente. Por ejemplo, cuando debemos comprender cómo funciona un código escrito por otra persona. Con esta guía se busca fomentar el uso de la depuración como una herramienta didáctica y como un complemento indispensable para el estudiante. Si bien en los primeros pasos, la falta de experiencia con el lenguaje y el compilador puede causar confusión, una vez dominadas las habilidades básicas de depuración se acortarán sensiblemente los tiempos de desarrollo y se podrá aprovechar al máximo la capacidad del IDE. Además, en proyectos de gran tamaño (como el proyecto final de la asignatura Programación Orientada a Objetos), en general resulta imposible realizar un seguimiento mental o una prueba de escritorio manual del programa, debido a la longitud y complejidad del código y a los múltiples caminos que su ejecución puede tomar.

En esta guía se desarrollarán tres ejemplos prácticos ilustrando el manejo básico de las facilidades de depuración, y un ejemplo adicional mostrando un caso especial que se presenta con frecuencia en la práctica. Cada uno de los primeros tres ejemplos debería realizarse como complemento a la práctica de una unidad de la asignatura. Es decir, el primer ejemplo corresponde a la práctica de la unidad 7 (estructuras de control), el segundo a la práctica de la unidad 8 (funciones), y el tercero a la practica de la unidad 9 (arreglos y structs). Para evitar mayores confusiones, se recomienda que no avance con ejemplos correspondientes a unidades que aún no han sido presentadas en las clases prácticas o teóricas, porque contendrán códigos y conceptos que tal vez generen confusión.

Consideraciones Previas

En el proceso de compilación, el código fuente se traduce a código de máquina. Sabemos que teniendo sólo el ejecutable (código de máquina) es imposible recuperar el código fuente. Pero más aún, teniendo ambos (ejecutable y fuentes) tampoco es trivial relacionarlos y saber por ejemplo, qué dirección de memoria le asignó el conjunto compilador+sistema operativo a una determinada variable, o qué conjunto de instrucciones de máquina se corresponden a una dada linea de código fuente. Es por esto que para que la depuración desde el código fuente sea posible, el compilador debe introducir dentro del ejecutable información adicional que le permita establecer luego esta relación. Por esto, este ejecutable (llamado comúnmente versión *Debug*) será más grande, y eventualmente más lento[3] que el ejecutable definitivo que el programador entregará al usuario final una vez concluido el proceso de desarrollo (denominado versión *Release*)[4]. Además, una vez compilado el ejecutable para depuración, se debe recordar que si se modifican los fuentes sin recompilar luego, se pierde la relación que existe entre ambos códigos, y en consecuencia el depurador podría mostrar información incorrecta.

Otro detalle a tener en cuenta, es que para poder controlar correctamente un programa, en la mayoría de los casos, el depurador debe ser el encargado de cargarlo en memoria y ejecutarlo, por lo que el IDE presentará generalmente dos formas de ejecución: la ejecución normal, y la ejecución a través del depurador.

3 La inclusión o no de la información de depuración no es la única diferencia entre las compilaciones Debug y Release. Hay otras diferencias (como niveles de optimización) que afectan de forma más perceptible la velocidad de ejecución del programa.

4 En Zinjal, los programas simples siempre se compilan en versión Debug, mientras que en los proyectos puede elegir la configuración desde el ítem “Opciones...” del menú “Ejecución”.

Ejemplo Nro 1: Control básico e inspección de variables

Paso 0: El programa ejemplo

Para llevar a cabo este ejemplo utilizaremos el siguiente código:

```
#include <iostream>
using namespace std;

int main(int argc, char *argv[]) {
     unitbuf(cout);
     int cant;
     cout << "Cantidad de datos: ";
     cin >> cant;
     double dato, sum = 0;
     for (int i=0; i<cant; i++) {
          cout << "Dato " << i << ": ";
          cin >> dato;
          sum += dato;
     }
     double prom = sum/cant;
     cout << "Promedio: " << prom << endl;
     return 0;
}
```

Cree un nuevo archivo utilizando la plantilla predeterminada y copie el código fuente del recuadro.

Este código corresponde a un programa que calcula y muestra el promedio de *n* números, donde *n* es un dato ingresado también por el usuario[5]. La variable *sum* acumula (suma) todos los enteros ingresados, la variable *cant* contiene la cantidad de enteros a ingresar, y finalmente, *prom* guardará el promedio (sum/cant). Dentro del bucle, *i* es el contador, y *n* es una variable auxiliar para leer los datos que se deben sumar en *sum*.

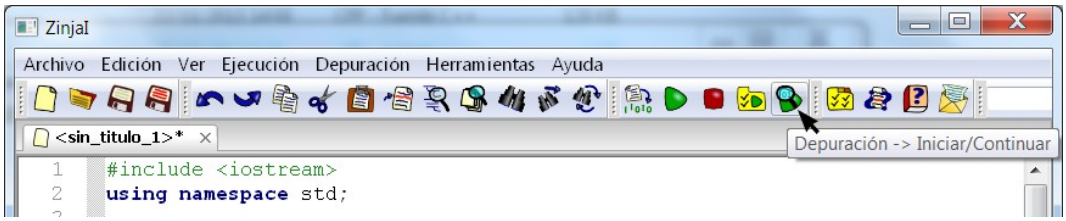
Paso 1: Detener el programa

Para que podamos inspeccionar las variables, o controlar el avance del programa, primero debemos hacer que se detenga en medio de la ejecución. Para esto, antes de comenzar a ejecutarlo, **debemos establecer *puntos de interrupción*** (breakpoints). Estos son puntos en el código (números de linea) donde el depurador debe detener el programa. Por lo general, se indican con un circulo rojo sobre el margen izquierdo. **Para colocarlo** en *Zinjal*, **puede hacer click sobre dicho margen**, o posicionar el cursor de texto en la linea de interés y presionar F8.

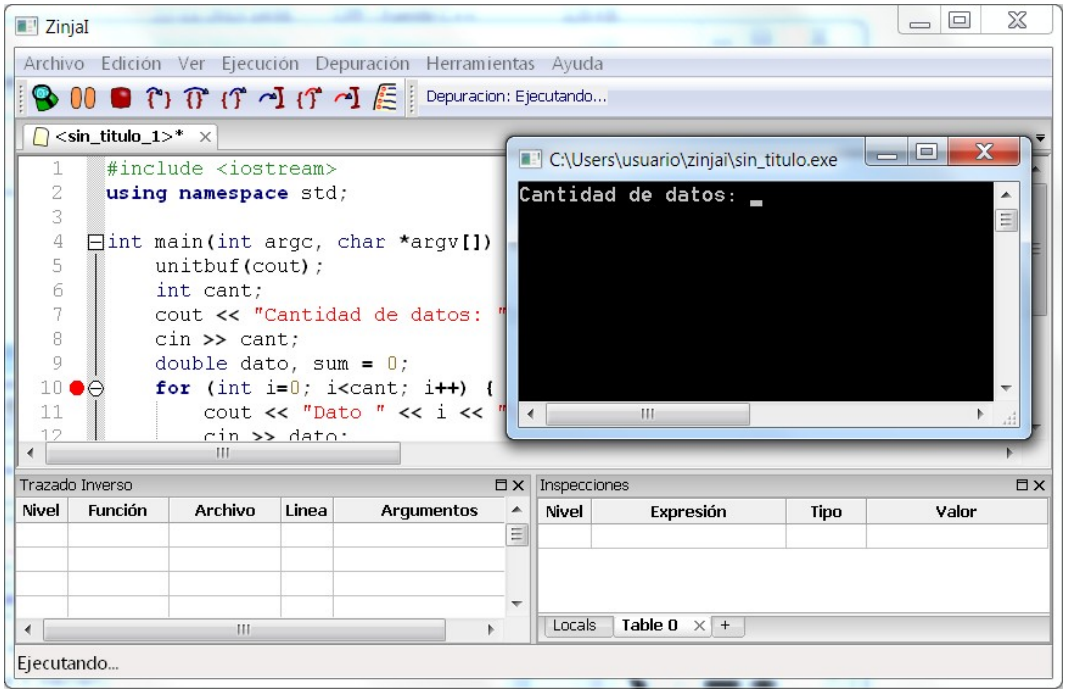
5 La linea que podría desconocer es la que dice "*unitbuf(cout)*;". Esta linea se utiliza para que *cout* muestre los mensajes en consola *inmediatamente*, ya que el comportamiento normal en algunos sistemas es que "espere" a acumular varios mensajes y los muestre todos juntos (a menos que reciba antes *endl* o *fllush)*. Este comportamiento podría confundir a la hora de ejecutar paso a paso el programa, ya que podrían no verse reflejadas inmediatamente en la consola, acciones sobre *cout* que ya hayan sido ejecutadas.

```
5       unitbuf(cout);
6       int cant;
7       cout << "Cantidad de datos: ";
8       cin >> cant;
9       double dato, sum = 0;
10      for (int i=0; i<cant; i++) {
11          cout << "Dato " << i << ": ";
12          cin >> dato;
13          sum = dato;
14      }
```

Pruebe colocar un punto de interrupción en la línea 10 (donde comienza el bucle *for*).

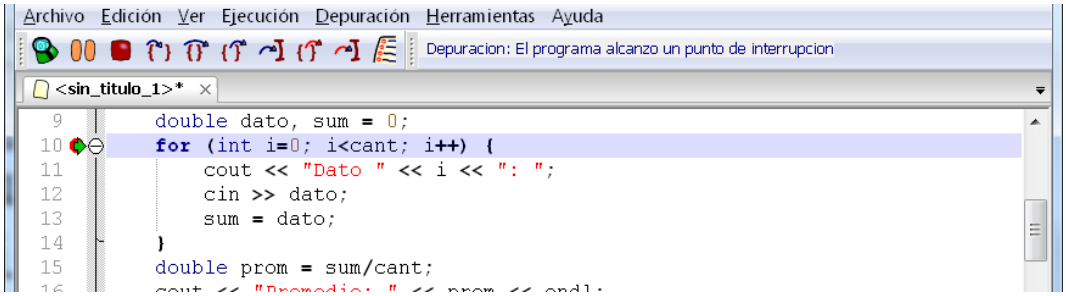


Una vez colocado el punto de interrupción, ejecute el programa con la tecla F5 (o la opción “Ejecutar” del menú “Depuración”). La ventana de *ZinjaI* desplegará dos nuevos paneles: el panel de *Trazado Inverso* y el panel de *Inspecciones* (más adelante ejemplificaremos su uso). Inmediatamente, el programa comenzará a correr normalmente[6] y le solicitará que ingrese el primer dato.



6 En GNU/Linux, algunas versiones de gdb muestran el mensaje “&”warning: GDB: Failed to set controlling terminal...”. Puede ignorarlo sin consecuencias. Avisa que algunas operaciones muy particulares sobre la terminal no funcionarán normalmente durante la depuración, pero en la práctica no necesitaremos de ninguna de esas operaciones.

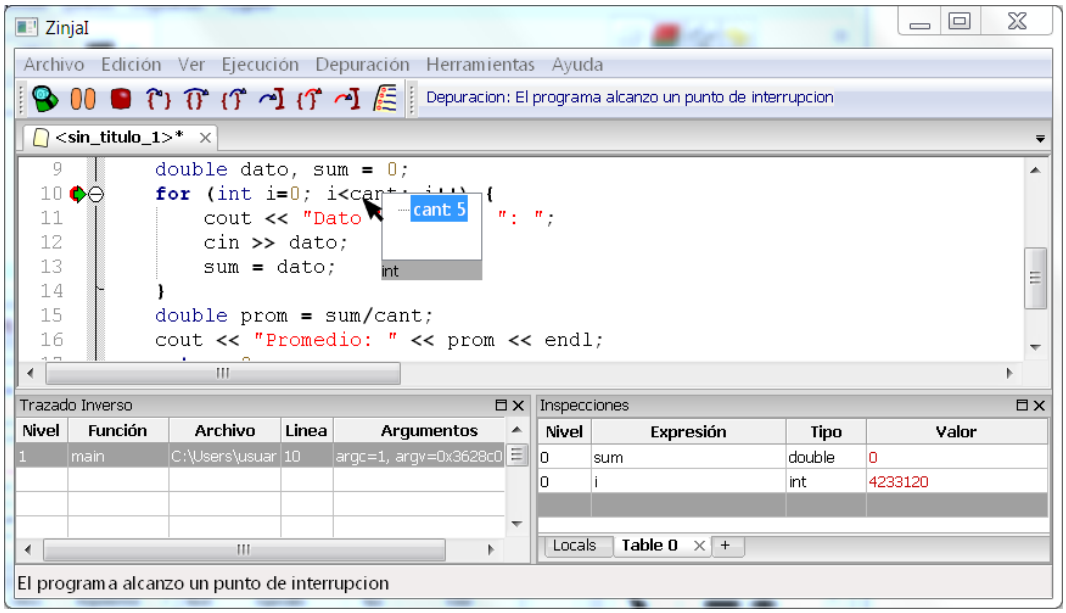
Ingrese 5 y presione *Enter*. Observará que el programa se detiene luego de leer el dato y la ventana de *Zinjal* pasa al frente (o parpadea en la barra de tareas). En el margen izquierdo del código encontrará una flecha verde sobre el punto de interrupción. **Esta flecha** indica dónde se ha detenido el programa. Cuando se marca una línea, **quiere decir que el programa se detuvo justo antes de ejecutar esa línea**.



Debe notar que no puede detener el programa en cualquier línea. Algunas, como comentarios, declaraciones de variables o lineas en blanco, no tienen correspondencia con ningún fragmento del ejecutable, por lo que no son puntos de interrupción válidos. Existen además otras formas de detener el programa sin colocar un punto de interrupción: una, por ejemplo, es pausarlo mientras se ejecuta con el botón de pausa de la barra de herramientas de *Zinjal*; otra es generando un error (por ejemplo, acceso a una posición de memoria inválida, o intentado resolver una división por cero).

Paso 2: Inspeccionar variables

Una vez que se ha detenido el programa, se pueden inspeccionar los contenidos de las variable. Existen tres formas básicas de hacerlo:



1. **Colocar el puntero del ratón sobre el nombre de la variable** (o seleccionar una expresión) **y esperar unos segundos** (en la figura se

muestra el valor de *cant*).

2. **Seleccionar la pestaña "Locals" en el "Panel de Inspecciones"** (ubicado abajo a la izquierda). Esta pestaña muestra todas las variables locales y sus valores actuales[7].
3. Seleccionar cualquier otra pestaña del "Panel de Inspecciones", e **ingresar la variable o expresión en alguna celda de la columna "Expresión"** (en la figura se muestran las variables *sum* e *i*).

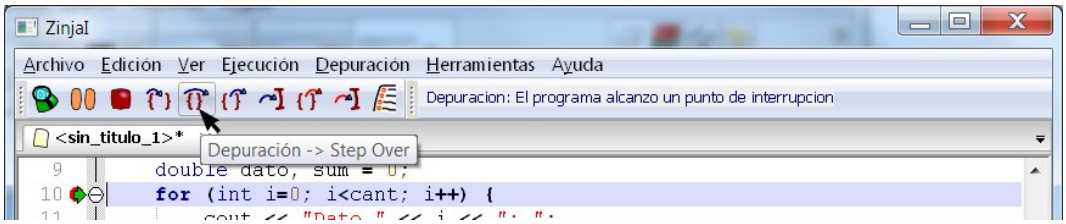
Observe que el programa se detuvo justo antes de comenzar el bucle, por lo que aún no inicializó el contador. A esto se debe el extraño valor de *i*.

Los dos primeros métodos son más rápidos, pero el tercero muestra más información (el tipo, el ambiente, etc.), y permite modificar el valor de una variable haciendo doble click sobre el mismo. Además, las entradas en la tabla de inspecciones permanecen allí cuando avanza el programa, por lo que en la próxima detención se actualizarán automáticamente mostrando los nuevos valores[8].

Paso 3: Continuar la ejecución

En este ejemplo, podemos continuar la ejecución de dos formas básicas. Una es avanzando un solo paso (una línea en el código); la otra es avanzando hasta el próximo punto de interrupción.

Para avanzar paso por paso, utilice la tecla F7 (*step over* en el menú "Depuración"). **Cada vez que presione F7 el programa avanza una línea de código.**



Para continuar ejecutando normalmente presione F5. Esta acción retomará la ejecución continua hasta que el programa alcance otro punto de interrupción. Si el programa no alcanzase nunca otro punto de interrupción se ejecutará hasta finalizar. En este caso, la consola de ejecución generalmente se cerrará automáticamente sin esperar a que presione una tecla, por lo cual es conveniente colocar un punto de interrupción en la última linea del *main* ("return 0;") para observar los resultados de la ejecución antes de que se cierre la ventana.

Pruebe continuar la ejecución paso por paso y observe cómo evolucionan las variables (recuerde que al llegar al paso que contiene el *cin* deberá volver a la consola de ejecución para ingresar un valor). En la tabla de inspecciones, las variables que cambien su valor se resaltarán en color rojo.

Para detener definitivamente el programa sin finalizar la ejecución presione Shift+F5 (o haga click en "Detener" en el menú "Depuración").

7 La pestaña "locals" podría mostrar también variables que aún no han sido inicializadas, o siquiera definidas según el código fuente, pero que pertenecerán al mismo scope/ámbito.

8 Se actualizan cada vez que el programa se pausa, no mientras este se ejecuta.

En todo momento, la parte derecha de la barra de herramientas le informa en letras azules el estado de la ejecución/depuración.

Ejemplo Nro 2: Funciones y ámbitos

Paso 0: El programa ejemplo

Para llevar a cabo este ejemplo utilizaremos el siguiente código:

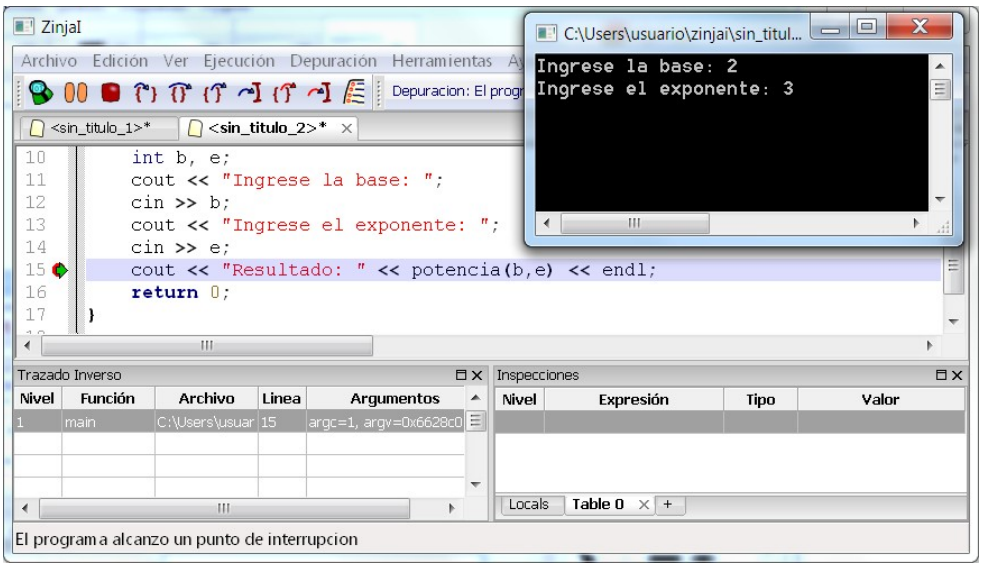
```
#include <iostream>
using namespace std;

int potencia(int base, int expo) {
     if (expo==0) return 1;
     else return base*potencia(base,expo-1);
}

int main(int argc, char *argv[]) {
     int b, e;
     cout << "Ingrese la base: ";
     cin >> b;
     cout << "Ingrese el exponente: ";
     cin >> e;
     cout << "Resultado: " << potencia(b,e);
     return 0;
}
```

Cree un nuevo archivo utilizando la plantilla predeterminada y copie el código fuente del recuadro.

Este código corresponde a un programa que calcula una potencia entera de un número entero. Para ello utiliza la forma recursiva $a^n = a \cdot a^{n-1}$.

Paso 1: Entender el trazado inverso

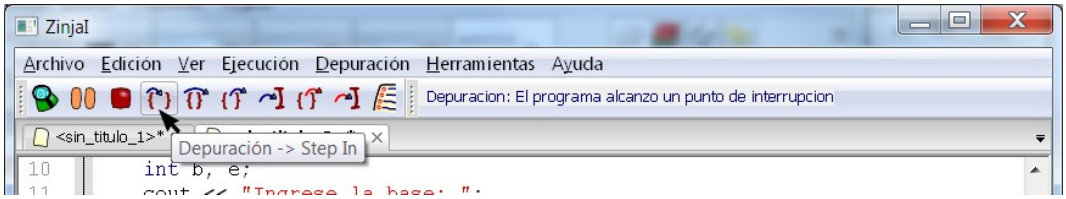
Coloque un punto de interrupción en la línea 15 (donde está la llamada "*potencia(b,e);*"), y presione F5 para ejecutar el programa hasta este punto. Cuando solicite base y exponente ingrese 2 y 3 respectivamente.



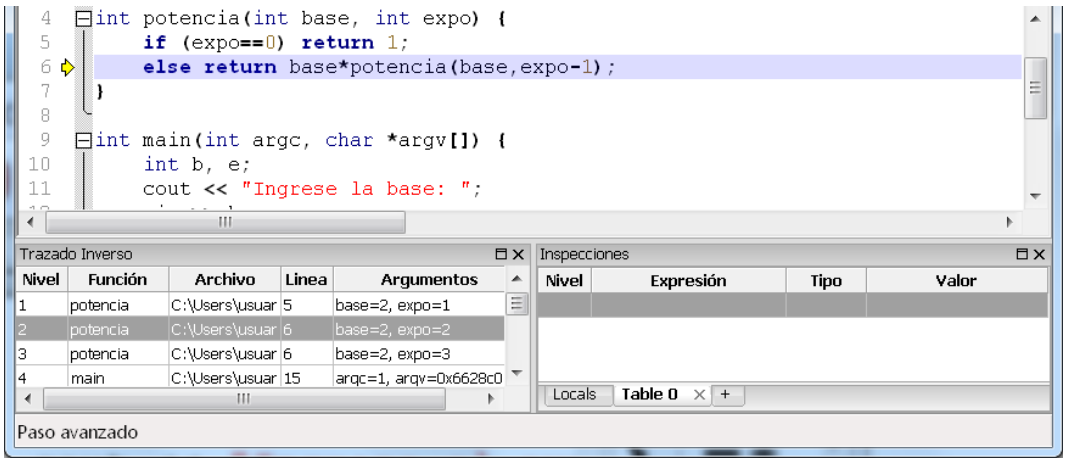
La ejecución debería detenerse justo antes de calcular el resultado. **La linea que corresponde ejecutar a continuación contiene la llamada a la función potencia**, por lo que en realidad, la próxima linea de código a ejecutar debería ser la primer línea de la función potencia (primero hay que evaluarla para luego poder mostrar el resultado). En estos casos, tenemos dos formas de continuar la ejecución: *step over* y *step in*.

La primera (*step over*, con la tecla F7) es la que utilizamos hasta ahora, **avanza a la siguiente linea dentro del ámbito actual**. En este caso, avanzará a la siguiente linea del *main*, realizando la evaluación de la función y mostrando el resultado en un solo paso.

La segunda forma de avanzar (*step in*, con F6), nos permite meternos dentro de la función. Es decir, si presionamos F6 el depurador avanzará hasta la primer línea de la función "potencia", y nos permitirá analizar qué ocurre dentro de la misma.



Presione F6 y observe el contenido del panel de trazado inverso (abajo a la derecha, *backtrace* en Inglés). En este panel tenemos en la primer linea la función actual y el punto donde realmente estamos detenidos en la ejecución, y en las otras lineas podemos observar cómo se llegó a este punto, y qué otras funciones estan "esperando" a que se resuelva esta llamada para poder continuar. En este caso, la primer línea indicará la función *potencia* y la segunda la función *main*. Puede hacer doble click sobre cada función para dirigirse a ese punto del código (observe que la flecha sobre el margen que indica la posición será verde cuando esté en el nivel más interno, y amarilla cuando se seleccione otro). Además, la tabla muestra una columna con los argumentos de la función. Presione F6 cuatro veces y observe como se "apilan" llamadas a *potencia*, con diferentes argumentos (recursividad).

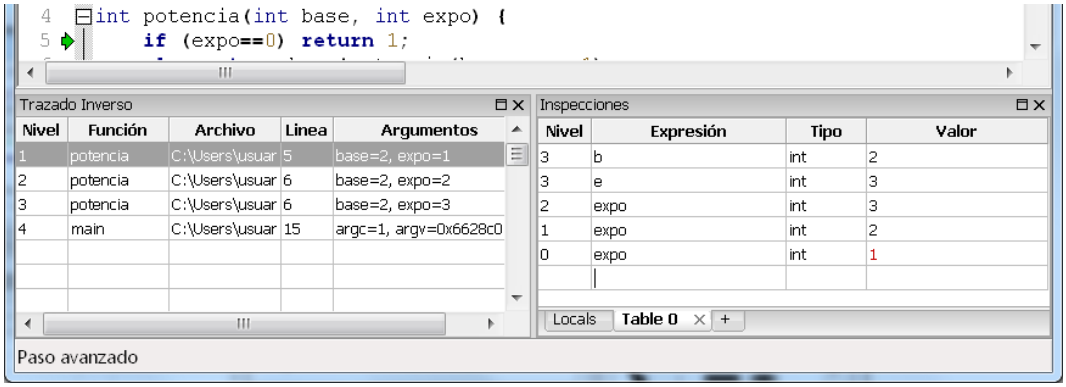


Paso 2: Identificar el ámbito de las inspecciones

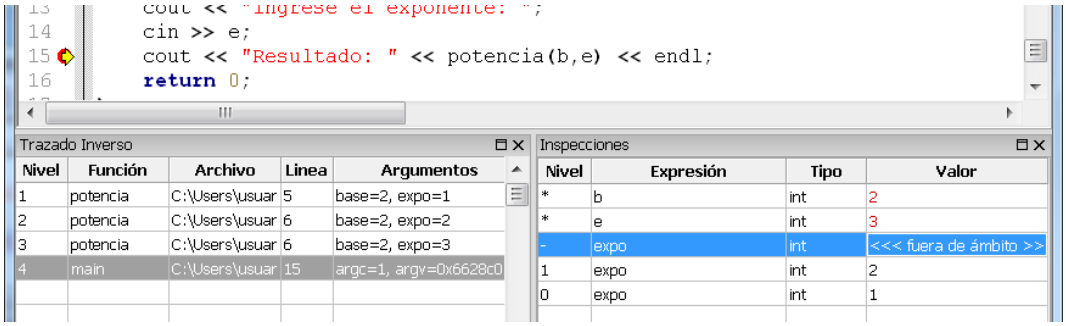
Ahora aparece un detalle más a tener en cuenta al usar la tabla de inspecciones. Se pueden encontrar distintas variables con el mismo nombre si estas están en distintos ámbitos (en este caso cada función define un ámbito o

scope). **Cuando se ingresa una inspección en la tabla, ésta se asocia al ámbito que se encuentre seleccionado** (el que marca la flecha verde/amarilla del margen). Se puede observar en la tabla de inspecciones una columna llamada “nivel” que indica en qué nivel (en relación a la tabla de trazado inverso) está el ámbito de esa inspección.

Pruebe inspeccionar la variable *expo* en distintos ámbitos (seleccione el ámbito con doble click en la tabla de trazado inverso, y luego ingrese la expresión “expo” en la tabla de inspecciones).

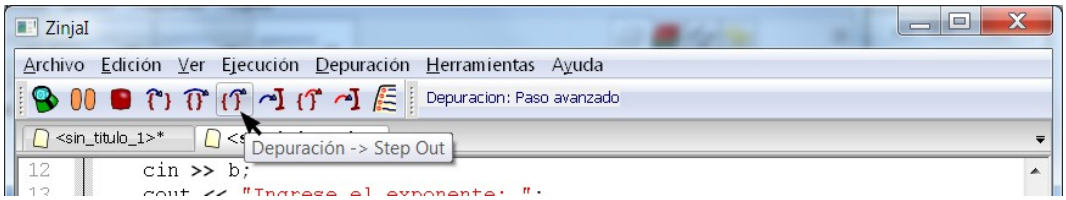


En *ZinjaI* existe una segunda forma de ver las inspecciones, en la cual las inspecciones no tienen un ámbito asociado, sino que se evalúan en el ámbito seleccionado, y al cambiar la selección, cambian su valor. Para estas expresiones, la columna “Nivel” contiene un asterisco (*) si son válidas para el ámbito actual, o un menos (-) si no lo son. Para convertir una expresión de un tipo en otra (es decir, asociar o desasociar con un ámbito), puede hacer doble click en la columna “Nivel”. Pruebe convertir una de las expresiones, y seleccionar diferentes funciones en el trazado inverso (incluyendo *main*, donde la variable no existe).

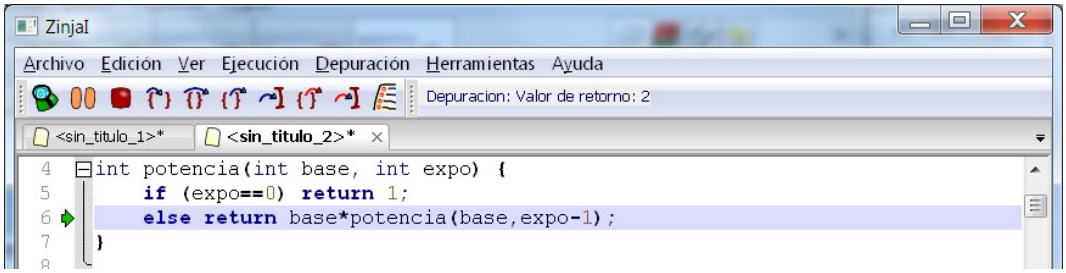


Paso 3: Continuar la ejecución

Finalmente, para continuar la ejecución, tenemos ahora nuevas alternativas. Una particularmente útil es indicarle al depurador que debe continuar ejecutando hasta salir del ámbito actual, es decir, hasta finalizar la función (siempre considerando la función del nivel 0). Para esto debe utilizar *step out* (Shift+F6).



Al utilizar esta acción, en la barra de estado de la depuración (el texto en azul a la derecha de la barra de herramientas) encontrará el valor de retorno que arrojó la función al finalizar. Pruebe este método seleccionando el nivel 0 en el trazado inverso y presionando Shift+F6. Observe que en el trazado inverso hay ahora un nivel menos, y en la parte superior izquierda de la pantalla se le indica que la función finalizó retornando el entero 2.



El cursor de ejecución queda posicionado en la llamada a la función que acaba de finalizar.

Ejemplo Nro 3: Visualización de estructuras de datos

Paso 0: El programa ejemplo

Para llevar a cabo este ejemplo utilizaremos el siguiente código:

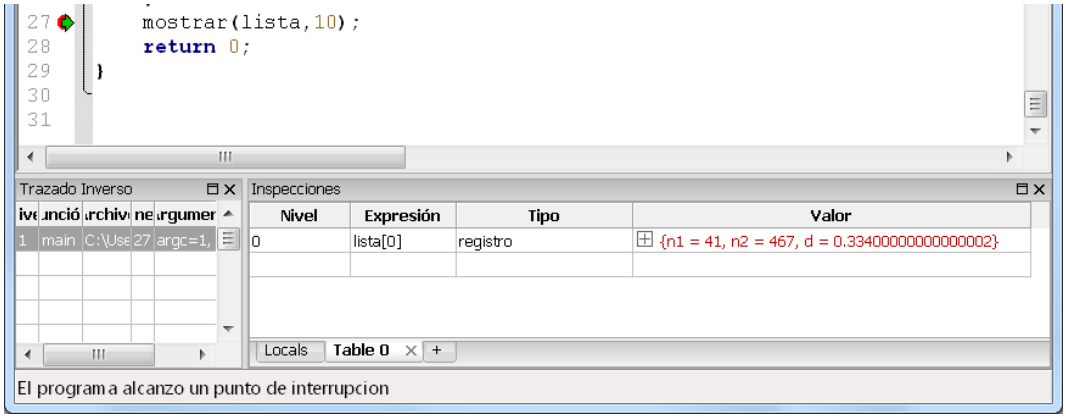
```
#include <iostream>
#include <iomanip>
#include <cstdlib>
using namespace std;

struct registro {
     int n1,n2;
     double d;
};

void mostrar(registro *arreglo, int n) {
     for (int i=0; i<n; i++)
          cout << I << right
               << setw(5) << arreglo[i].n1 << "  "
               << setw(4) << arreglo[i].n2 << "  "
               << arreglo[i].d << endl;
}

int main(int argc, char *argv[]) {
     registro lista[10];
     for (int i=0; i<10; i++) {
          lista[i].n1 = rand()%1000;
          lista[i].n2 = rand()%1000;
          lista[i].d = double(rand()%1000)/1000;
     }
     mostrar(lista,10);
     return 0;
}
```

Cree un nuevo archivo utilizando la plantilla predeterminada y copie el código fuente del recuadro.

El programa declara una estructura *registro* con tres campos (dos enteros *n1 y n2,* y un real *d*), una función *mostrar* que imprime en pantalla un arreglo de registros y una función *main* que genera un arreglo estático de 10 registros aleatorios, y luego lo muestra utilizando la función *mostrar*.

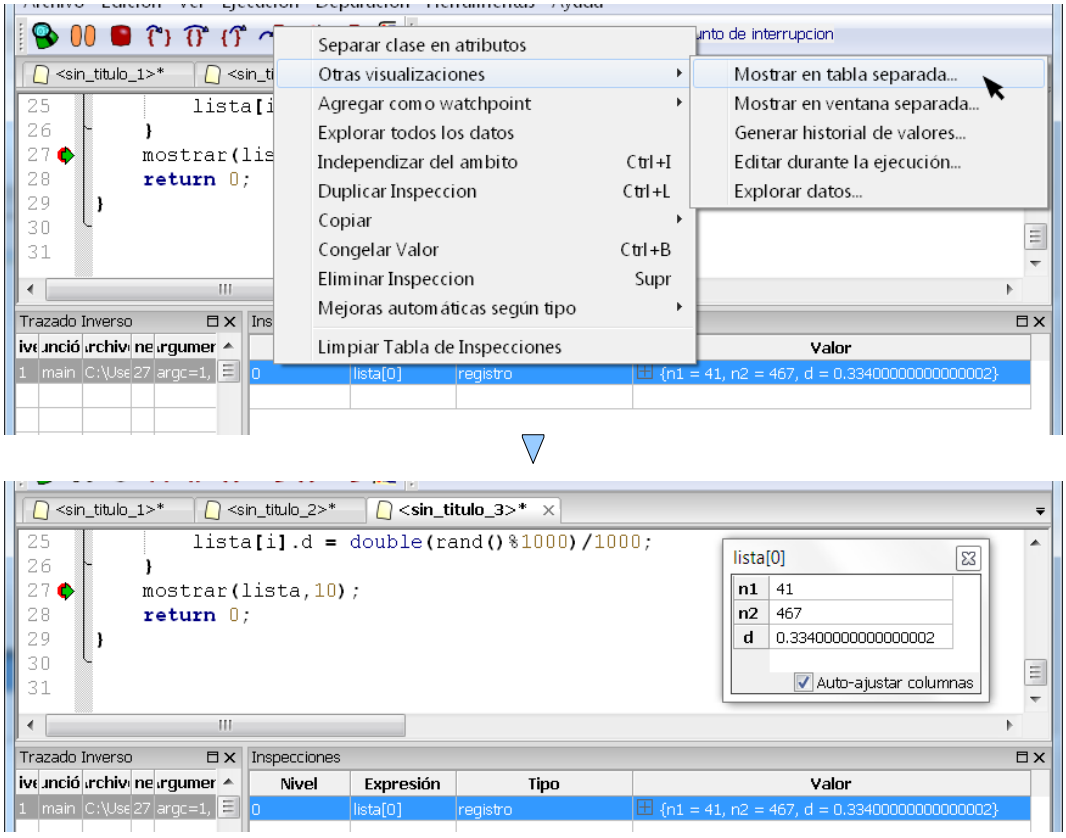
Paso 1: Inspección de estructuras

Coloque un punto de interrupción en la linea 27 (la llamada a la función *mostrar*) y presione F5 para ejecutar hasta ese punto. Allí tendremos cargados diez registros aleatorios. Ingrese la inspección “lista[0]” en la tabla de inspecciones para observar el contenido del primer registro.

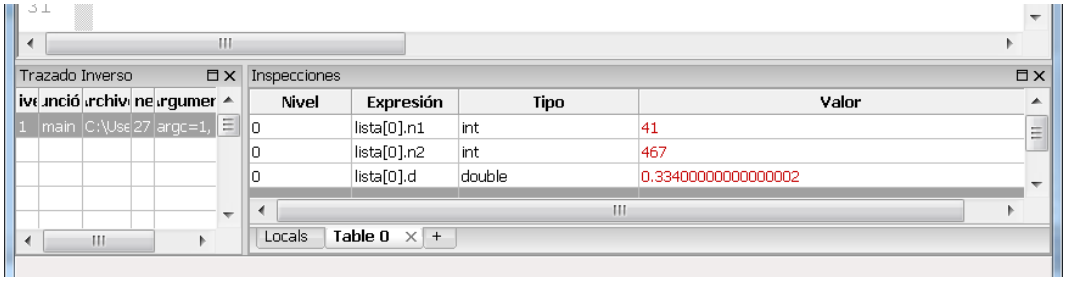


El valor de la inspección contiene los tres campos separados por comas. Hay dos formas de observarlas con mayor comodidad:

1) Haga click sobre la misma con el botón derecho del ratón y seleccione en el menú contextual “Otras visualizaciones” y luego “Mostrar en tabla separada”. Se desplegará una nueva ventana con la estructura desplegada en una tabla.



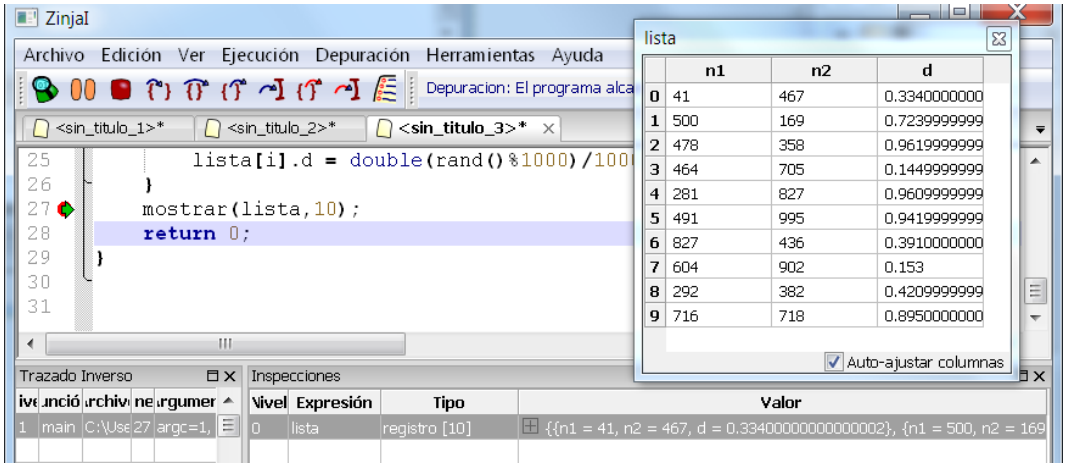
2) Haga doble click sobre la celda con el valor de la inspección en la tabla de inspecciones. Esta acción separa una inspección compuesta en múltiples inspecciones más simples (un struct en sus atributos); y puede realizarse siempre que la inspección muestre el símbolo ⊞ precediendo su valor.



Paso 2: Inspección de arreglos

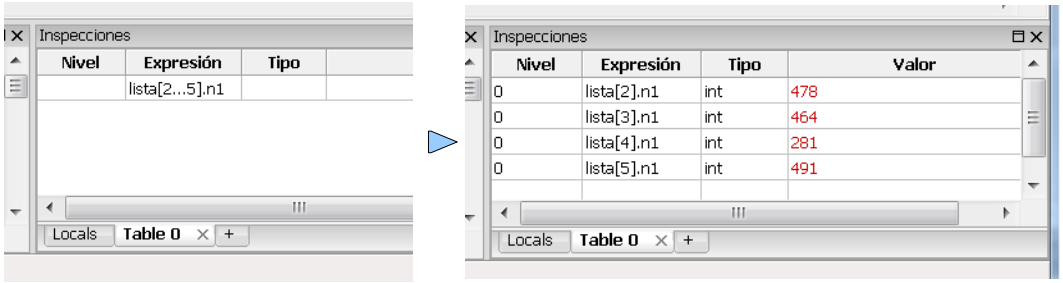
Borre las inspecciones de la tabla (seleccionándolas y presionando la tecla delete/suprimir, o con la opción “Limpiar tabla de inspecciones” del menú contextual.

Ingrese ahora la inspección “lista” y observe como se muestra toda la lista en un solo renglón. De forma similar a las estructura registro del paso 1, puede hacer doble click para “desarmar” la inspección en diez inspecciones (una por elemento), o seleccionar la opción “Mostrar en tabla separada” del menú contextual para visualizarla en una nueva ventana.

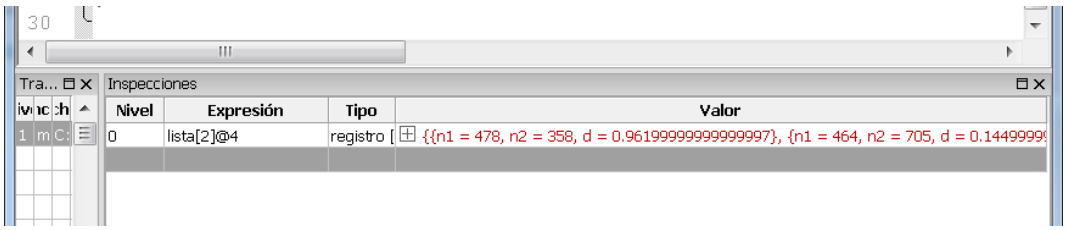


Además de estas dos opciones, existen dos alternativas más para visualizar solo una parte del arreglo, por ejemplo desde los índices 2 a 5 (cuatro elementos), ingresando una notación especial. Para ello puede introducir alguna de las siguientes expresiones:

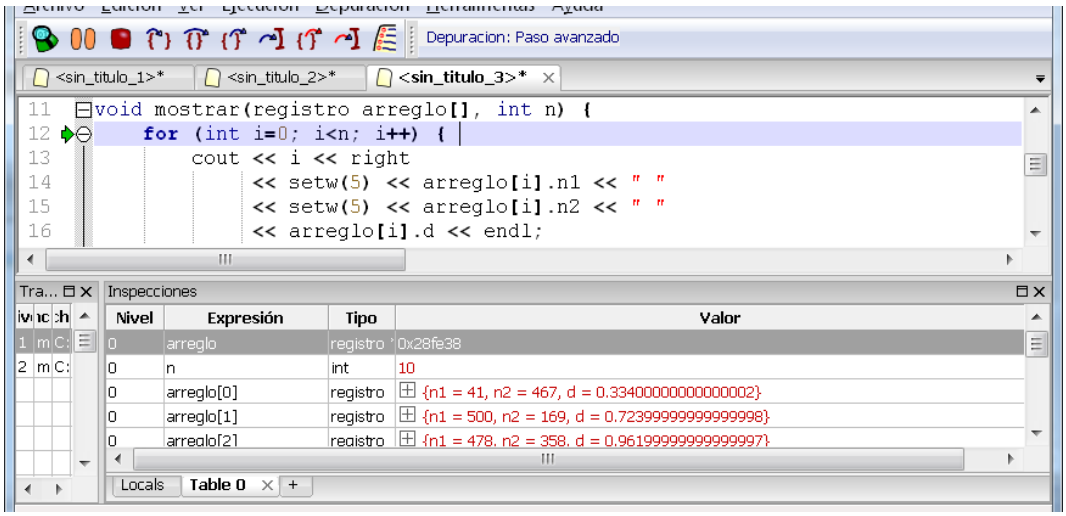
1) “lista[2...5].n1”: el uso de tres puntos suspensivos generará automáticamente 4 inspecciones variando el índice entre 2 y 5 (inclusive).



2) “lista[2]@4”: el uso de la arroba significa que se deben mostrar 4 variables ubicadas en la memoria en forma contigua (como ocurre con los arreglos lineales). A diferencia del primer método, de esta forma se genera una sola inspección con la lista de cuatro elementos, que además puede visualizarse luego en una tabla separada.



Finalmente, presione F6 para ingresar en la función *mostrar* y una vez dentro ingrese la inspección “arreglo”. Observe que en este caso no vemos los elementos, sino una dirección de memoria (la del primer elemento, donde comienza el arreglo). Esto se debe a que dentro de la función, es imposible saber el tamaño del arreglo a partir del mismo (por eso necesitamos el parámetro *cant*). Sin embargo, aún puede mostrar sus elementos ya sea como inspecciones individuales (utilizando opcionalmente “@” o “...”) o en tabla separada (se le solicitará ingresar la longitud del arreglo al abrir la tabla).



Resumen de comandos

Se resumen a continuación los comandos presentados para controlar la ejecución:

- Colocar/quitar punto de interrupción (F8)
-  Iniciar depuración o continuar ejecución (F5)
-  Avanzar a la siguiente línea, *step over* (F7)
-  Ingresar en la función, *step in* (F6)
-  Salir de la función, *step out* (Shift+F6)
-  Ejecutar hasta donde se encuentra el cursor (Shift+F7)

Existen otras posibilidades no desarrolladas en esta guía, principalmente destinadas a alterar la ejecución (como salir de una función sin ejecutar lo que falta, continuar ejecutando desde un punto arbitrario, deshacer la ejecución (solo en *GNU/Linux*), etc.). Una vez familiarizado con el depurador, puede investigar por su cuenta estas funcionalidades.

Conclusiones

Ya habrá observado que las herramientas de depuración pueden ser de gran ayuda a la hora de encontrar errores. Sin embargo, estas herramientas no señalan el problema, sino que sólo se limitan a exponer ante el programador la evolución del programa.

Cuando el programa se interrumpe por un error en tiempo de ejecución, la ejecución con el depurador generalmente mostrará el punto exacto en donde se manifestó el problema, y permitirá inspeccionar el estado del programa para intentar deducir cómo se llegó a la situación problemática[9].

Cuando los errores son de lógica (el programa finaliza, pero el resultado no es el esperado), es el programador quien debe saber dónde colocar los puntos de interrupción y qué variables inspeccionar para poder encontrar el problema. Por lo general, tendrá un sospecha o intuición acerca del fragmento de código que causa un problema, y realizará pruebas con datos simples de modo que pueda comparar sus respuestas calculadas con lápiz y papel con los valores observados en las inspecciones en cada paso.

Poder encontrar el lugar y las variables adecuadas, aunque no es trivial al principio, es una “habilidad” que se mejora con la práctica, y que requiere un buen conocimiento del lenguaje y del problema por parte del programador.

Finalmente, aún en un código “sin errores”, estas herramientas pueden ser muy útiles a la hora de analizar el flujo de datos o la estructura del programa para entender su funcionamiento.

9 Cuando un programa se detiene inesperadamente durante la ejecución normal, el paso más lógico consiste en reproducir la ejecución con el depurador (ingresando las mismas entradas) para determinar exactamente dónde se interrumpe. El estudiante debería adquirir este comportamiento como una acción refleja, ya que es el primer paso en el análisis del error, y en muchos casos el problema se hará evidente simplemente inspeccionando el trazado inverso y las variables locales de cada función en ese estado.