

## **Práctica 1. Depuración y Análisis de la eficiencia.**

### **Comparación de la eficiencia de algoritmos de ordenación de vectores**

Esta práctica tiene dos objetivos: aprender a depurar un programa, y diseñar un programa que permita comparar la eficiencia de distintos algoritmos de ordenación de vectores.

#### **Parte 1. Depuración**

Diremos que un programa es correcto cuando no hay *errores de compilación* (es decir, el programa compila) y tampoco hay *errores de ejecución* (es decir, el programa hace lo que tiene que hacer). Los errores de compilación suelen ser más fáciles de subsanar ya que el compilador nos informa de dichos errores y nos indica la línea donde se han producido. Por el contrario, descubrir porqué se ha producido un error de ejecución puede ser más complicado ya que supone inspeccionar nuestro código en busca de errores. Para facilitar esta tarea, una herramienta fundamental es el depurador (o en inglés, *debugger*).

Depurar un programa significa someterlo a un ambiente de ejecución controlado por medio de herramientas dedicadas a ello. Este ambiente permite, entre otras cosas, conocer exactamente el flujo de ejecución del programa, el valor que las variables adquieren, y la pila de llamadas a funciones. La depuración de un programa es útil cuando el programa no muestra los resultados que se esperan para cierta entrada de datos debido a que el programador cometió algún error durante el proceso de diseño. Muchas veces encontrar este tipo de fallos suele ser difícil, ya sea porque la percepción del programador no permite encontrar el fallo en su diseño o porque la errata es muy pequeña. En este caso es de mucha utilidad conocer paso a paso cómo se ejecutan las estructuras de control, que valor adquieren las variables, etc.


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.
- Punto de ruptura (en 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, por ejemplo el valor de las variables en determinados momentos del programa. También sirve para verificar hasta que punto el programa se ejecuta sin problemas.
- 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.
- Visualizar el valor de las variables: permite conocer el valor de alguna o varias variables.

A continuación veremos cómo utilizar el depurador de CodeBlocks.

1. Crea un nuevo proyecto del tipo “Console Application” en CodeBlocks.
2. Descarga del aula virtual el fichero depurador.cpp y copia dicho código al fichero main.cpp de tu proyecto. En el fichero depurador.cpp se ha definido la función `mayorDeCuatro` que debería calcular el máximo de 4 números.
3. Compila y ejecuta el programa. Como podrás observar, no se obtiene el resultado esperado. A continuación veremos como depurar el programa.
4. Como hemos comentado anteriormente los puntos de ruptura sirven para pausar tu programa en cierta instrucción (línea de código). Para añadir un punto de ruptura, selecciona primero la línea de código donde quieres parar. Ahora haz clic en el menú Depurar y luego en Añadir/Quitar Punto de Ruptura (si los menús de CodeBlocks están en inglés esta opción está en el menú *Debug*, la opción *Toggle Breakpoint*), o pulsando la tecla F5. Observa que aparece un octógono rojo en la línea donde has añadido el punto de ruptura. Haciendo clic en la barra en el margen izquierdo del editor a la altura de la línea tendrá el mismo efecto que los pasos anteriores. Pon un punto de ruptura en la línea 9 del programa, es decir donde se inician los valores de las variables `a`, `b`, `c`, y `d`.

```
1  #include <iostream>
2
3  using namespace std;
4
5  int mayorDeCuatro(int x1, int x2, int x3, int x4);
6
7  int main()
8  {
9      int a=10, b=5, c=15, d=21;
10     int mayor = mayorDeCuatro(a,b,c,d);
11     cout << "El mayor elemento entre "<<a<< ", "<<b << ", "<<c<< " y "<<d << " es: " << mayor<<endl;
12     return 0;
13 }
14
15 int mayorDeCuatro(int x1, int x2, int x3, int x4){
16     int mayor1, mayor2, mayor;
17     if (x1 > x2){
18         mayor1 = x1;
19     }
20     else{
21         mayor1 = x3;
22     }
23     if (x3 > x4){
24         mayor2 = x3;
25     }
26     else{
27         mayor2 = x2;
28     }
29     if (mayor1 > mayor2){
30         mayor = mayor2;
31     }
32     else{
33         mayor = mayor1;
34     }
35     return mayor;
36 }
37
38 }
```

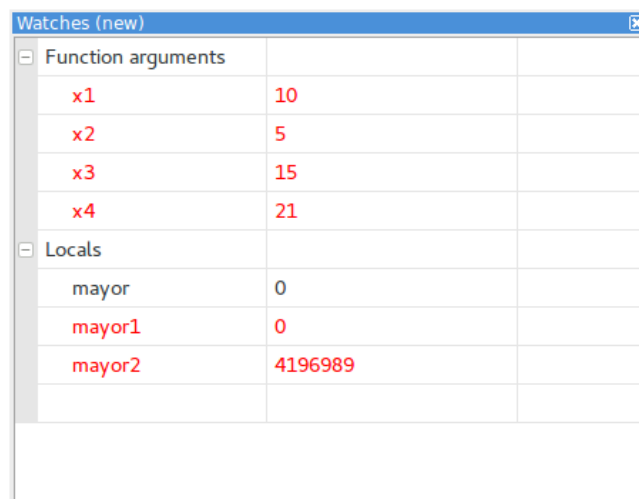
5. Ahora vamos a ejecutar nuestro programa con el depurador. Para ello ve al menú Depurar y selecciona la opción Empezar (en inglés *Debug* → *Start/Continue*), o pulsa la tecla F8, o pulsa la flecha roja: 

Cuando carga el depurador si tu programa ejecuta la línea de código con el punto de ruptura, CodeBlocks te avisará de que ha llegado a esa línea poniendo una flecha amarilla sobre el punto de ruptura. Dicha flecha amarilla nos indica en que paso de nuestro programa se encuentra el depurador.

```
8  {
9      int a=10, b=5, c=15, d=21;
10     int mayor = mayorDeCuatro(a,b,c,d);
11     cout << "El mayor elemento entre "<<a<< ", "<<b << ", "<<c<< " y "<<d << " es: " << mayor<<endl;
12     return 0;
13 }
14 }
```

Ahora puedes examinar los datos en tu programa o ejecutarlo paso a paso.

6. Una vez que hemos llegado a un punto de ruptura, puedes saltar pasos en tu programa de las distintas formas siguientes (todas estas maneras de interactuar con el depurador están disponibles en el menú *Debug* y en los iconos al lado de la flecha roja):
- Siguiente línea (*next line*), tecla F7. El depurador dará un paso a la siguiente instrucción (línea de código) en tu programa. Utiliza esta funcionalidad para avanzar a la línea 10 de tu código.
  - Avanzar paso a paso (*step into*), combinación Shift+F7. El depurador saltará una instrucción si esa instrucción es una llamada a una función saltará a ella (está es la diferencia con el avance “Siguiente línea”). Utiliza esta funcionalidad para avanzar de nuevo en la línea 10, verás que en lugar de continuar con el *main* el depurador, ahora la flecha de ejecución del depurador ha pasado al método *mayorDeCuatro*.
- Existen otras opciones, pero las dos anteriores son las más comunes.
7. Una de las grandes ventajas de la depuración es que podemos mostrar el valor de las variables observadas en un determinado instante. Puedes observar tus variables de dos formas distintas:
- Colocando el ratón sobre una variable en tu código fuente. Para ello hay que tener activada la opción *Evaluate Expression under cursor* que se encuentra en el menú *Settings* → *Debugger* → *Default*.
  - Habilitando la ventana *watches* desde el menú *Debug* → *Debugging windows* → *Watches*.



Watches (new)	
Function arguments	
x1	10
x2	5
x3	15
x4	21
Locals	
mayor	0
mayor1	0
mayor2	4196989

La ventaja de esta segunda opción es que desde la ventana *Watches* podemos realizar operaciones con las variables que se están observando en ese momento. Prueba a ponerte en la última celda de la ventana *Watches* y escribe  $x1+x2$ , verás que se muestra la suma de las variables *x1* y *x2*.

Para poder ver el contenido de un vector y no únicamente su dirección de memoria, hay que pulsar sobre él con el botón derecho, pulsar en *Properties*, activar la casilla *Watch as an array* y seleccionar el rango.

**Ejercicio.** Utilizando el depurador, avanza paso a paso por las instrucciones de la función *mayorDeCuatro* y comprueba los valores que van tomando las distintas variables. A continuación encuentra y soluciona los errores de dicha función.

**Ejercicio.** Modifica el `main` para leer los valores de las variables `a`, `b`, `c`, y `d` desde el teclado utilizando la instrucción `cin`. Si ahora repites el proceso de depuración verás que el depurador se queda “atascado” en la primera instrucción de `cin`. Lo que ha hecho el programa es abrir una ventana de DOS y está esperando a que introduzcas el valor que debe leer. Localiza esa ventana e introduce el valor, verás como el depurador avanza a la siguiente instrucción.

## Parte 2. Comparación de la eficiencia

En la segunda parte de la práctica vamos a comparar la eficiencia de distintos algoritmos de ordenación. Los algoritmos de ordenación con los que vamos a trabajar son: *selección directa*, *inserción directa*, *burbuja* y *heapsort*. Al ejecutar el programa debe aparecer el siguiente menú:

1. Selección directa
2. Inserción directa
3. Burbuja
4. Heapsort
5. Salida

que permitirá seleccionar el método de ordenación. En el Campus Virtual encontrarás el fichero `EsqueletoP1.cpp`. Este fichero contiene los distintos algoritmos de ordenación nombrados anteriormente, pero los métodos de selección directa e inserción directa están mal implementados, así que nuestra primera tarea consistirá en solucionar los errores utilizando el depurador.

**Nota importante:** pese a que los vectores en C++ almacenan datos desde la componente 0, en esta práctica ignoraremos el valor de esta componente por no perder legibilidad en algunos de los algoritmos (ya que en caso contrario, es decir, comenzando en la componente 0, la traducción no es totalmente automática). Por lo tanto, los datos del vector a ordenar estarán almacenados de la componente 1 en adelante.

Los pasos a seguir para completar esta parte de la práctica son los siguientes:

1. Crea un nuevo proyecto del tipo “Console Application” en CodeBlocks.
2. En el fichero `main.cpp` copia el código del fichero `EsqueletoP1.cpp` que encontrarás en el aula virtual.
3. Comprueba que entiendes lo que hace el `main`, y las funciones `crearVector` y `escribirVector`.
4. Ejecuta el programa, crea un vector de tamaño 5 y comprueba que con el método de selección directa el programa no termina, y con el método de inserción directa el resultado no es el esperado. Comprueba también que el resto de métodos funcionan de manera correcta.
5. Utilizando el depurador, encuentra los errores de las funciones `seleccionDirecta` e `insercionDirecta`. Recuerda que para ver el contenido de un vector y no únicamente su dirección de memoria, hay que pulsar sobre él con el botón derecho, pulsar en *Properties*, activar la casilla *Watch as an array* y seleccionar el rango.
6. Vamos ahora a medir el tiempo invertido en ordenar el vector. Para calcular el tiempo de ejecución utilizaremos la función `clock_t clock()` que está definida en el fichero de cabecera `<time.h>`. Esta función mide intervalos de tiempo dentro de una única ejecución. Para calibrar el resultado que devuelve la función `clock()` se utiliza la macro `CLOCKS_PER_SEC`. Así la medida de dos instantes `t1` y `t2` nos permitirá medir el tiempo transcurrido en segundos mediante la expresión `double (t2-t1) / CLOCKS_PER_SEC`.

Así, una forma de calcular el tiempo de ejecución de una llamada a un subprograma de ordenación es la siguiente:

```
clock_t t1, t2;  
...  
t1=clock();  
...           //Llamada a la acción ordenar  
t2=clock();  
...  
cout<<double(t2-t1)/CLOCKS_PER_SEC<<endl;
```

Por último, calcula el orden exacto de los distintos algoritmos de ordenación y comprueba con distintos tamaños del vector (1000, 50000, 100000, 500000, ...) si los tiempos de ejecución crecen de acuerdo con los resultados teóricos. Para ello rellena la siguiente tabla:

	Selección directa	Inserción directa	Burbuja	HeapSort
n = 1000	seg	seg	seg	seg
n = 10.000	seg	seg	seg	seg
n = 50.000	seg	seg	seg	seg
n = 100.000	seg	seg	seg	seg
n = 500.000	seg	seg	seg	seg
n = 1.000.000	seg	seg	seg	seg