

Entrada / Salida rápida en C, C++ y Java

Agustín Santiago Gutiérrez

Facultad de Ciencias Exactas y Naturales
Universidad de Buenos Aires

Training Camp 2016

Contenidos

1 Contexto

2 C

3 C++

4 Java

“Pero la velocidad era poder, y la velocidad era gozo, y la velocidad era pura belleza.”

Richard Bach, “Juan Salvador Gaviota”

Contenidos

1 Contexto

2 C

3 C++

4 Java

¿Por qué conviene hacer eficiente la E/S?

- En problemas de complejidad lineal o similar, las operaciones de E/S pueden insumir un porcentaje importante del tiempo total de ejecución, que es lo que se mide en la mayoría de las competencias.

¿Por qué conviene hacer eficiente la E/S?

- En problemas de complejidad lineal o similar, las operaciones de E/S pueden insumir un porcentaje importante del tiempo total de ejecución, que es lo que se mide en la mayoría de las competencias.
- Aún si los tiempos elegidos por el jurado son generosos, y es posible con una solución esperada resolver el problema aún con mecanismos de E/S ineficientes, usar formas eficientes de hacer E/S nos permitirá siempre “zafar” con programas más lentos que si no lo hiciéramos así.

¿Por qué conviene hacer eficiente la E/S?

- En problemas de complejidad lineal o similar, las operaciones de E/S pueden insumir un porcentaje importante del tiempo total de ejecución, que es lo que se mide en la mayoría de las competencias.
- Aún si los tiempos elegidos por el jurado son generosos, y es posible con una solución esperada resolver el problema aún con mecanismos de E/S ineficientes, usar formas eficientes de hacer E/S nos permitirá siempre “zafar” con programas más lentos que si no lo hiciéramos así.
- Existen diferencias **muy simples y pequeñas** en la forma de realizar E/S en los programas, que **generan grandes diferencias** en el tiempo total insumido por estas operaciones. Conocer estas diferencias es entonces obtener un beneficio relevante con muy poco esfuerzo.

Contenidos

1 Contexto

2 C

3 C++

4 Java

Funciones printf y scanf

- En C plano, la forma de E/S más utilizada son las funciones printf y scanf. Estas funciones **son eficientes**, y es la forma recomendada de realizar entrada salida en este lenguaje.
- Ejemplo:

```
#include <stdio.h>
int main() {
    int x,y;
    scanf("%d%d", &x, &y);
    printf("%d\n", x+y);
}
```

Contenidos

1 Contexto

2 C

3 C++

4 Java

Funciones printf y scanf

- En C++, las mismas funciones scanf y printf siguen disponibles, y siguen siendo una opción eficiente para aquellos que estén acostumbrados o gusten de usarlas.
- Ejemplo:

```
#include <cstdio>
using namespace std;
int main() {
    int x,y;
    scanf("%d%d", &x, &y);
    printf("%d\n", x+y);
}
```

Streams cin y cout

- La forma elegante de hacer E/S en C++ es mediante los streams cin y cout (Y análogos objetos fstream si hubiera que manipular archivos específicos en alguna competencia).
- Ejemplo:

```
#include <cstdio>
using namespace std;
int main() {
    int x,y;
    cin >> x >> y;
    cout << x+y << endl;
}
```

Por defecto en casos usuales, cin y cout son lentos

- La eficiencia relativa de cin y cout vs scanf y printf dependerá del compilador y arquitectura en cuestión.
- Dicho esto, en la mayoría de los compiladores y sistemas usuales utilizados en competencia, cin y cout son por defecto **mucho** más lentos que scanf y printf.
- Veremos algunos trucos para que cin y cout funcionen más rápido. Con ellos, en algunos sistemas comunes funcionan más rápido que printf y scanf, pero la diferencia es muy pequeña.
- En otras palabras, aplicando los trucos que veremos a continuación, da igual usar cin y cout o printf y scanf, ambas son eficientes.

Primera observación: endl

- El valor “endl” no es solo un fin de línea, sino que además ordena que se realice un **flush del buffer**.
- De esta forma, imprimir muchas líneas cortas (un solo entero, un solo valor Y/N, etc) realiza muchas llamadas a escribir directamente al sistema operativo, para escribir unos poquitos bytes en cada una.
- **Solución:** utilizar `\n` en su lugar. Esto es un sencillo caracter de fin de línea, que no ejecuta un flush del buffer.
- Ejemplo:

```
#include <cstdio>
using namespace std;
int main() {
    int x,y;
    cin >> x >> y;
    cout << x+y << "\n";
}
```

Segunda observación: sincronización con stdio

- Por defecto, `cin` y `cout` están sincronizados con todas las funciones de `stdio` (notablemente, `scanf` y `printf`). Esto significa que si usamos ambos métodos, las cosas se leen y escriben en el orden correcto.
- En varios de los compiladores usuales esto vuelve a `cin/cout` **mucho** más lentos, y si solamente usamos `cin` y `cout` pero **nunca** `scanf` y `printf`, no lo necesitamos.
- **Solución:** utilizar `ios::sync_with_stdio(false)` al iniciar el programa, para desactivar esta sincronización. Notar que si hacemos esto, **ya no podemos usar `printf` ni `scanf`** (ni ninguna función de `stdio`) sin tener resultados imprevisibles.
- Desactivar la sincronización también puede tener efectos al utilizar más de un thread. Esto no nos importa en ICPC.

Segunda observación: sincronización (ejemplo)

Esta optimización tiene efectos muy notorios, típicamente reduce el tiempo de ejecución a la mitad en varios jueces online comunes.

Ejemplo:

```
#include <cstdio>
using namespace std;
int main() {
    ios::sync_with_stdio(false);
    int x,y;
    cin >> x >> y;
    cout << x+y << "\n";
}
```


Tercera observación: dependencia entre cin y cout

- Por defecto, cin está *atado* a cout, lo cual significa que siempre antes de leer de cin, se fuerza un flush de cout. Esto hace que programas interactivos funcionen como se espera.
- Cuando solo se hacen unas pocas escrituras con el resultado al final de toda la ejecución, esto no tiene un efecto tan grande.
- Si por cada línea que leemos escribimos una en la salida, este comportamiento fuerza un flush en cada línea, como hacía endl.
- **Solución:** utilizar `cin.tie(NULL)` al iniciar el programa, para desactivar esta dependencia. Notar que si hacemos esto, tendremos que realizar flush de cout manualmente si queremos un programa interactivo.

Tercera observación: dependencia (ejemplo)

```
#include <cstdio>
using namespace std;
int main() {
    ios::sync_with_stdio(false);
    cin.tie(NULL);
    int x,y;
    cin >> x >> y;
    cout << x+y << "\n";
}
```

Ejemplo final con las 3 técnicas

- Eliminar sincronización con stdio
- Eliminar dependencia entre cin y cout
- No utilizar endl

```
#include <cstdio>
using namespace std;
int main() {
    ios::sync_with_stdio(false);
    cin.tie(NULL);
    int x,y;
    cin >> x >> y;
    cout << x+y << "\n";
}
```

Contenidos

1 Contexto

2 C

3 C++

4 Java

InputStreams, OutputStreams, Readers, Writers

- En Java existe la distinción entre los Streams (bytes) y los Readers / Writers (caracteres unicode).
- Aún siendo todo ASCII, para archivos de texto uno termina trabajando siempre con readers y writers porque tienen las funciones más cómodas.
- El “análogo” de cin y cout en Java es System.in y System.out.
- Sin embargo, hay que tener cierto cuidado ya que al operar con ellos directamente, no se bufferean las operaciones, y tenemos un problema de permanente flushing, similar al que ocurría en C++ con endl.
- Particularmente, hacer `System.out.println(x)` es exactamente como `cout << x << endl`, y queremos evitarlo.

Ejemplo típico de I/O con Java

```
import java.io.*;
import java.util.*;

class HelloWorld {
    public static void main(String [] args) throws Exception {
        Scanner scanner = new Scanner(System.in);
        int n = scanner.nextInt();
        long total = 0;
        for (int i = 0; i < n; i++) {
            long x = scanner.nextLong();
            total += x;
            System.out.println(total);
        }
    }
}
```

Esto es lento, porque no usa buffers, lee y escribe directamente.

Introduciendo Buffers

```
import java.io.*;
import java.util.*;

class HelloWorld {
    public static void main(String [] args) throws Exception {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        BufferedWriter bw = new BufferedWriter(new OutputStreamWriter(System.out));

        Scanner scanner = new Scanner(br);
        PrintWriter printer = new PrintWriter(bw);
        int n = scanner.nextInt();
        long total = 0;
        for (int i = 0; i < n; i++) {
            long x = scanner.nextLong();
            total += x;
            printer.println(total);
        }
        printer.flush();
    }
}
```

¡¡Notar el flush!! No se puede omitir. Al usar buffers, `printer.println` no imprime en el momento, y sin flush al final pueden quedar cosas pendientes de escribir en la salida (se observa una salida "cortada").

En versiones nuevas de Java...

```
import java.io.*;
import java.util.*;

class HelloWorld {
    public static void main(String [] args) throws Exception {
        Scanner scanner      = new Scanner(System.in);
        PrintWriter printer = new PrintWriter(System.out);
        int n = scanner.nextInt();
        long total = 0;
        for (int i = 0; i < n; i++) {
            long x = scanner.nextLong();
            total += x;
            printer.println(total);
        }
        printer.flush();
    }
}
```

En versiones nuevas, esto “zafaría”, gracias a que Scanner y PrintWriter usan buffers internos. Notar que usar System.out y System.in directamente sin envolverlos nunca usan buffers.

No obstante, la versión anterior es la jugada segura todo terreno. Si el rendimiento de E/S puede importar, **siempre usar buffers**.

Más eficientes, pero más incómodos

Podemos evitar por completo `PrintWriter` y `Scanner` y resolver todo con `BufferedWriter` y `BufferedReader`:

```
import java.io.*;
import java.util.*;

class HelloWorld {
    public static void main(String [] args) throws Exception {
        BufferedReader br = new BufferedReader(new InputStreamReader(System.in));
        BufferedWriter bw = new BufferedWriter(new OutputStreamWriter(System.out));

        int n = Integer.valueOf(br.readLine());;
        long total = 0;
        for (int i = 0; i < n; i++) {
            long x = Long.valueOf(br.readLine());;
            total += x;
            bw.write(String.valueOf(total));
            bw.newLine();
        }
        bw.flush();
    }
}
```

La diferencia entre `PrintWriter` y `BufferedWriter` no es muy grande (En casos como el ejemplo, < 10 %).

La diferencia entre `Scanner` y `BufferedReader` es potencialmente muy grande (puede ser un 50 %). Otra función a evitar en estos casos es `String.split`, que es bastante lenta.