Departamento de
Informática e Ingeniería
de Sistemas
Universidad Zaragoza

Práctica 2

Tecnología de Programación
Grado en Ingeniería Informática

2026-02-08

Objetivos

- Aprovechar los mecanismos que proporciona la herencia para maximizar la reutilización de código.
- Diseñar jerarquías de herencia que permitan polimorfismo por inclusión.
- Utilizar algunos de los contenedores proporcionados por las bibliotecas estándar de C++ y Java.

Esta práctica la realizarás **tanto en C++ como en Java**.

1. Máquina de pila

Para el desarrollo de un nuevo **microcontrolador de tipo pila**, se te pide que implementes una máquina virtual que emule dicho controlador y que permita experimentar con el juego de instrucciones y hacer análisis de eficiencia sobre el mismo.

Los programas de nuestro microcontrolador usan como memoria de trabajo una única pila, y todas las órdenes de su juego de instrucciones están diseñadas para tomar sus operandos de la pila (desapilándolos con instrucciones `'pop'`) y para devolver sus resultados en la propia pila (apilándolos con instrucciones `'push'`). El microcontrolador trabaja únicamente con enteros de 32 bits (tipo `int`) y por tanto todas las operaciones trabajarán con dicho tipo de datos. La ejecución de cada programa usa una pila independiente, para facilitar la multitarea.

El microcontrolador en cuestión consta de un juego de **instrucciones** muy pequeño pero potente, que permite desarrollar una gran variedad de **programas**. Iremos presentando las instrucciones específicas a lo largo del guión de la práctica.

2. Implementación de la máquina virtual

Deberás implementar, en ambos lenguajes de programación (C++ y Java), la **máquina virtual** que ejecuta los programas de nuestro microcontrolador, con todo el juego de instrucciones propuesto. Te recomendamos que el desarrollo de dicha máquina virtual sea progresivo, siguiendo el guión de esta práctica.

En primer lugar se implementarán las piezas fundamentales de la máquina de pila:

- **Pila:** No es necesario implementar una pila de cero. Puedes utilizar estructuras de datos de las bibliotecas estándar del lenguaje concreto. En C++ dispones de `std::stack<int>`[1], y en Java de `Stack<Integer>`[2]. Ambas estructuras de datos disponen de una función para introducir un elemento en la pila (`push( )` en los dos lenguajes), para mirar la cima de la pila (`top( )` en C++ y `peek( )` en Java) y para eliminar la cima de la pila (`pop( )` en ambos lenguajes).

- **Instrucción:** Deberá ser capaz de representar una instrucción del juego de instrucciones del microcontrolador. Cada una de las instrucciones tiene un comportamiento diferente sobre la pila al ser ejecutadas. Dicho comportamiento específico deberás implementarlo mediante *polimorfismo por inclusión*,

[1] `https://en.cppreference.com/w/cpp/container/stack`
[2] `https://docs.oracle.com/en/java/javase/25/docs/api/java.base/java/util/Stack.html`

representando cada instrucción concreta como una clase derivada de una clase base dentro de una jerarquía de herencia. Las clases derivadas serán cada una de las instrucciones del juego de instrucciones, y tendrán diferente comportamiento. Específicamente, una instrucción es algo que puede ejecutarse, y al ejecutarse modifica la pila (cada instrucción en particular, de forma diferente). El contador de instrucciones (PC, *program counter*) avanza para determinar la siguiente instrucción del programa a ejecutar (normalmente la siguiente, pero puede depender de la instrucción concreta como veremos más adelante).
Adicionalmente, para poder listar el programa, cada instrucción deberá poder devolver una cadena de texto que represente la propia instrucción.

- **Programa:** Deberá representar un programa. El programa deberá contener un vector de instrucciones. Es necesario que sea un vector porque se debe poder acceder a las instrucciones, indexadas por el contador de instrucciones, que se usa para recorrer dicho programa. Desconoces a priori el número de instrucciones máximas de un programa, así que necesitarás que dicho vector esté en memoria dinámica, y, como es natural, cada programa puede tener un número distinto de instrucciones. Para esto puedes utilizar los mecanismos clásicos del lenguaje para manejar vectores en memoria dinámica. El vector deberá poder almacenar instrucciones de cualquier tipo, por lo que en C++ será necesario manejar explícitamente *punteros* a la clase base `Instruccion`, para poder disponer de polimorfismo por inclusión mediante herencia:

  ```
  Instruccion** instrucciones;
  ```

  o bien, usando un tipo adicional, para aclarar la sintaxis con múltiples `'*'`:

  ```
  using InstruccionPtr = Instruccion*;
  ...
  InstruccionPtr* instrucciones;
  ```

  Un programa deberá poder:

  - Ser **ejecutado**, empezando cada ejecución con una pila vacía y con el contador de programa a `0` (en la primera instrucción). El programa finalizará en cuanto el contador de programa sobrepase la última instrucción del programa (lo que habitualmente ocurrirá cuando su valor salga fuera del rango válido del índice del vector).
  - Ser **listado**, mostrando por la salida estándar (pantalla) todas las instrucciones del programa junto a su número de línea (el que tiene el contador del programa cuando la ejecuta).

  El constructor del programa por defecto genera un vector de instrucciones completamente vacío. Para definir un programa concreto (en lugar de cargarlo de un fichero de texto, lo cual es tedioso y está fuera de las necesidades académicas de esta práctica) deberás hacerlo mediante herencia. La herencia en este caso no sirve para implementar polimorfismo, sino exclusivamente para la reutilización de código: se reutiliza el código para la ejecución y el listado, y cada clase derivada de tu `Programa` base implementará un programa determinado: es el constructor de cada programa concreto el que define el código del programa insertando las instrucciones concretas en el vector de instrucciones. En el caso de C++, el destructor del programa será el encargado de liberar toda la memoria que use (teniendo en cuenta que en el constructor se ha reservado memoria dinámica tanto para el vector de instrucciones como para cada instrucción en concreto). En Java no es necesario (ni existe) destructor, dado que la memoria se libera a través del mecanismo de recolección de basura de la máquina virtual.

Una vez representados todos los elementos fundamentales de la máquina virtual, en los siguientes apartados y progresivamente, irás definiendo nuevos programas y añadiendo nuevas instrucciones al juego de instrucciones implementado.

3. Primer programa: suma de dos números

El primer programa que vas a definir sobre esta máquina de pila es un programa muy sencillo que calcula la suma de dos números introducidos por teclado, y muestra el resultado por pantalla.

Para este programa, deberás añadir las siguientes instrucciones al juego de instrucciones (como se ha explicado antes, mediante herencia con respecto a la clase base que representa una instrucción):

add	suma los dos elementos de la cima de la pila: desapila dos valores de la pila y apila su suma
read	lee un valor y lo apila: pide un valor al usuario por la entrada estándar, lo lee y lo apila (indicando al usuario de alguna forma que espera una entrada de datos)
write	muestra un valor y lo elimina: desapila un valor de la pila y lo escribe en la salida estándar

Para el propio programa deberás definir una clase nueva que herede de la clase `Programa` y que en su constructor genere el vector con las instrucciones correspondientes, en el orden adecuado.

El listado del programa que deberá obtenerse será el siguiente:

```
0   read
1   read
2   add
3   write
```

Si has definido las clases base según lo indicado en el apartado anterior, te darás cuenta de que para añadir las tres clases nuevas (que representan las tres nuevas instrucciones de nuestro juego de instrucciones) y el nuevo programa, no has tenido que modificar ninguna parte del código previo.

Para probar el correcto funcionamiento de todo, implementa un programa principal `main.cc` que simule la ejecución de la máquina virtual, es decir, que cree el programa, lo liste y lo ejecute. La interacción con el programa principal deberá ser como la siguiente (en negrita lo que teclea el usuario):

```
~> main ⏎
Programa:
0   read
1   read
2   add
3   write

Ejecucion:
? 3 ⏎
? 4 ⏎
7
```

4. Cuenta atrás

Este segundo programa deberá pedirle al usuario un número, que asumiremos que es entero y positivo[3] y mostrará por pantalla una cuenta atrás desde ese número, de uno en uno, hasta llegar a cero. Para este segundo programa necesitarás las siguientes instrucciones nuevas:

[3] Es decir, no hace falta que hagas ningún tipo de comprobación.

push <c>	apila un valor: apila la constante <c> (parámetro de la instrucción) en la pila
dup	duplica la cima de la pila: desapila un valor y lo reapila dos veces
jumpif <l>	salto condicional a la instrucción <l>: desapila la cima de la pila, y si su valor es mayor o igual que cero salta la ejecución del programa a la instrucción <l> (parámetro de la instrucción) en caso contrario, la ejecución continúa en la siguiente instrucción del programa

El listado del programa que deberá obtenerse será el siguiente:

```
0   read
1   dup
2   write
3   push -1
4   add
5   dup
6   jumpif 1
```

Este programa te obliga a considerar dos aspectos nuevos con respecto al juego de instrucciones del microcontrolador:

- El primero es cómo representar instrucciones que tienen parámetros, tales como `push` y `jumpif`. Esos parámetros forman parte de la instrucción: ¿ Dónde se almacenan ? ¿ Cómo se le pasa el valor correspondiente a la instrucción ?
- El segundo es la posibilidad de tener saltos: la instrucción `jumpif` puede generar un salto. Un salto produce una modificación en el contador de programa, de tal forma que ya no se incrementa en uno para indicar la siguiente instrucción a ejecutar, sino que se le asigna un valor determinado.

Aunque ya ha sido mencionado de pasada en el apartado 2, esto es un cambio con respecto a las instrucciones que has implementado hasta ahora, en las que siempre se modificaba el contador de programa incrementándolo en uno, para pasar a la siguiente instrucción del programa. Es posible que no hayas tenido en cuenta en tu solución inicial la posibilidad de que una instrucción modifique el contador de programa de forma arbitraria, lo cual puede que te obligue a rediseñar la clase base de las instrucciones y el proceso de ejecución de un programa. Antes de hacer la modificación, considera que cualquier modificación a la clase base puede, potencialmente, obligar a modificar todas las clases derivadas (siendo que la mayoría de las instrucciones avanzan el contador de programa en uno). Definiendo los métodos acecuados puedes llegar a conseguir este comportamiento particular del salto en el `jumpif` sin tener que modificar las clases derivadas que ya existen.

Modifica el programa principal del apartado anterior para probar también este nuevo programa, a continuación del anterior. La interacción con el programa debería incluir la siguiente parte, correspondiente al nuevo programa:

```
Programa:
0   read
1   dup
2   write
3   push -1
4   add
5   dup
6   jumpif 1

Ejecucion:
? 5 ↵
5
4
3
2
1
0
```

5. Factorial

El último de los programas pedirá al usuario un número y calculará su factorial. Para ello necesitarás definir nuevas instrucciones para tu microcontrolador:

`mul`	multiplica los dos elementos de la cima de la pila: desapila dos valores de la pila y apila su producto
`swap`	intercambia los dos elementos de la cima de la pila: desapila dos valores y los reapila en orden inverso
`over`	duplica el elemento que está antes que el de la cima de la pila: desapila dos valores, apila el segundo desapilado, después el primero y por último de nuevo el segundo

Con estas nuevas instrucciones, define un nuevo programa, cuyo listado deberá ser el siguiente:

```
 0   push 1
 1   read
 2   swap
 3   over
 4   mul
 5   swap
 6   push -1
 7   add
 8   dup
 9   push -2
10   add
11   jumpif 2
12   swap
13   write
```

En esta última tarea es importante destacar que el desarrollo de las nuevas instrucciones y del programa, te ha costado bastante poco (sobre todo si lo comparas con soluciones alternativas a la herencia). Esto es un buen indicativo de la ventaja que tiene este mecanismo de los lenguajes orientados a objetos en términos de escalabilidad (¿cómo de fácil es extender el programa para soluciones más complejas?) y mantenibilidad (¿cómo de sencillo es localizar y hacer modificaciones?).

Modifica el programa principal del apartado anterior para probar también este nuevo programa, a continuación de los dos anteriores. La interacción con el programa debería incluir la siguiente parte, correspondiente al nuevo programa:

```
Programa:
 0   push 1
 1   read
 2   swap
 3   over
 4   mul
 5   swap
 6   push -1
 7   add
 8   dup
 9   push -2
10   add
11   jumpif 2
12   swap
13   write

Ejecucion:
? 6 ↵
720
```

Entrega

Asegúrate que tu programa principal se encuentra en el fichero **`main.cc`** en C++ o **`Main.java`** para Java. Tu programa principal deberá crear un programa de cada uno de los tres tipos, siguiendo el orden del guión de la práctica, y para cada uno de ellos listarlo y ejecutarlo. Además del comportamiento que te pedimos explícitamente, puedes definir todos los métodos y clases que consideres necesarios. Se valorará especialmente la escalabilidad del sistema, como por ejemplo la posibilidad de añadir instrucciones al juego de instrucciones, sin tener que modificar fragmentos de código común a todas ellas. También se valorará que no exista código repetido o métodos innecesarios.

Los archivos de código fuente de la práctica deberán estar organizados en dos subdirectorios `'c++'` y `'java'`, cada uno de ellos conteniendo todos los ficheros con el el código fuente de tu solución en el lenguaje correspondiente, siguiendo la siguiente estructura:

```
practica2_XXXXXX_XXXXXX
     \---- c++
          \---- Makefile
          \---- main.cc
          \---- instruccion.h
          \---- instruccion.cc
          \---- ...
     \---- java
          \---- Main.java
          \---- Instruccion.java
          \---- ...
```

Deberás entregar todos los archivos de código fuente que hayas necesitado para resolver el problema. Adicionalmente, para C++, deberás incluir un archivo *Makefile* que se encargue de compilar todos tus archivos de código fuente (**.cc*) y generar el ejecutable.

La solución en C++, deberá ser compilable y ejecutable con los archivos que has entregado y con el *main.cc* que has desarrollado, mediante los siguientes comandos:

```
make
./main
```

La solución en Java deberá ser compilable y ejecutable con los archivos que has entregado y con el *Main.java* que hayas desarrollado, mediante los siguientes comandos:

```
javac Main.java
java Main
```

No se deben utilizar paquetes, bibliotecas, ni ninguna infraestructura adicional fuera de las bibliotecas estándar del propio lenguaje.

En caso de no compilar siguiendo estas instrucciones, **la nota de la evaluación de la práctica será un `0`**.

Todos los archivos de código fuente (y directorios, si es el caso) solicitados en este guión deberán ser comprimidos en un único archivo ZIP con el siguiente nombre:

- **`practica2_<nip1>_<nip2>.zip`** (donde `<nip1>` y `<nip2>` son los NIPs de los estudiantes involucrados) si el trabajo ha sido realizado en pareja.
  En este caso sólo uno de los dos estudiantes deberá hacer la entrega en Moodle.
- **`practica2_<nip>.zip`** (donde `<nip>` es el NIP del estudiante involucrado) si el trabajo ha sido realizado de forma individual.

El archivo comprimido a entregar no debe contener ningún fichero aparte de los fuentes (y en su caso el fichero `Makefile`) que te pedimos: ningún fichero ejecutable (`*.exe`) u objeto (`*.o`), clases compiladas de Java (`*.class`), ficheros de interfaz de Haskell (`*.hi`), archivos de configuración del entorno de desarrollo (`.vscode`), ni ningún otro fichero adicional.

La entrega del archivo ZIP deberá hacerse en la tarea correspondiente preparada en el curso Moodle de la asignatura y antes de la fecha límite fijada para ello.