Departamento de
Informática e Ingeniería
de Sistemas
Universidad Zaragoza
1542

Práctica 2

Tecnología de Programación
Grado en Ingeniería Informática

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 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.

Un microcontrolador de pila consta como memoria a corto plazo de una única pila, y todas las instrucciones del juego de instrucciones están diseñadas para coger sus operandos de la pila (desapilándolos, *pop*) y para devolver su resultado en la propia pila (apilándolos, *push*). El microcontrolador trabaja únicamente con enteros de 32 bits (tipo `int`) y por tanto todas las operaciones trabajan con ese tipo de datos.

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 desarrollar, en el lenguaje elegido, la **máquina virtual** con todo el juego de instrucciones propuesto. Sin embargo, te recomendamos que el desarrollo de dicha máquina virtual sea progresivo, siguiendo el guión de esta práctica.

Lo primero es desarrollar 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:** Deberás de ser capaz de representar una instrucción. La particularidad es que cada una de las instrucciones tiene un comportamiento diferente sobre la pila al ser ejecutadas. Ese comportamiento diferente deberás representarlo mediante polimorfismo por inclusión, representando la instrucción como una clase base de una jerarquía de herencia. Las clases hijas serán cada una de las instrucciones del juego de instrucciones, y tendrán diferente comportamiento. Específicamente, una instrucción es algo

[1] `https://en.cppreference.com/w/cpp/container/stack`
[2] `https://docs.oracle.com/en/java/javase/17/docs/api/java.base/java/util/Stack.html`

que puede ejecutarse, y al ejecutarse modifica la pila (cada instrucción particular de forma diferente), y avanza en 1 (o en otro valor, dependiendo de la instrucción) el contador de instrucciones, para indicar la siguiente instrucción a ejecutar. Adicionalmente, para poder listar el programa, cada instrucción deberá poder devolver una cadena de texto que represente la propia instrucción.

- **Programa:** Deberás representar un programa. El programa deberá contener un vector de instrucciones. Necesitas que sea un vector porque debes poder acceder a instrucciones concretas indexadas según el contador de instrucciones (que es el que recorre el 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. En Java esto es sencillo (todos los vectores se crean de forma dinámica), pero en C++ necesitarás un campo adicional que indique el número de instrucciones, y además el vector será de *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

  - **Ejecutarse**, empezando 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 se quede fuera del rango válido del vector (lo cual, habitualmente, ocurrirá por sobrepasar la última instrucción del programa).
  - **Listarse**, 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 dejará el 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 polimorfismo sino exclusivamente para reutilización de código: se reutiliza la ejecución y el listado, y cada clase derivada de tu programa implementa un programa concreto: es el constructor de cada clase hija del programa el que rellena las instrucciones concretas. En el caso de C++, además, el destructor el programa será el encargado de liberar toda la memoria (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 para el juego de instrucciones.

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 una clase base para instrucciones):

- `add` - desapila dos valores de la pila y apila su suma.
- `read` - pide un valor al usuario por la entrada estándar (indicando al usuario de alguna forma que espera una entrada de datos) y lo apila en la pila.
- `write` - desapila un valor de la pila y lo muestra por la salida estándar (pantalla) al usuario.

Para el propio programa deberás definir una clase nueva que herede de la clase que representa un programa y que en su constructor rellene las instrucciones correspondientes en el orden adecuado. El listado del programa (que devolverá esta nueva clase gracias a la reutilización de código) será:

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

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

Para probar el correcto funcionamiento de todo, implemente un programa principal `main.cc` que cree el programa, lo liste y lo ejectute. La interacción con el programa debería ser como la siguiente:

```
~> 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 positivo y entero 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:

- `push <c>` - apila la constante <c> (parámetro de la instrucción) en la pila.
- `dup` - duplica la cima de la pila (desapila su valor y lo reapila dos veces).
- `jumpif <l>` - desapila la cima de la pila, y si su valor es mayor o igual que cero salta la ejecución del programa a la línea <l> (parámetro de la instrucción).

(continúa en la siguiente página...)

El programa de la cuenta atrás deberá mostrar el siguiente listado:

```
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 a las instrucciones:

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

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 la 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 hijas (siendo que la mayoría de las instrucciones avanzan el contador de programa en uno). Si le das ciertas vueltas podrías llegar a conseguir el no tener que modificar las clases hijas y conseguir este comportamiento particular del salto en el `jumpif`.

Modifica el programa principal del apartado anterior para probar este nuevo programa, además del anterior. La interacción con el programa debería incluir la 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 le pide al usuario un número y calcula su factorial. Para ello necesitas nuevas instrucciones del juego de instrucciones:

- `mul` - desapila dos valores de la pila y apila su producto.

- swap - intercambia dos elementos en la cima de la pila (desapila dos valores y los reapila en orden inverso).
- over - copia (apila) el valor que está tras la cima de la pila (desapila dos valores, apila el segundo desapilado, después el primero y por último el segundo otra vez).

Con estas nuevas instrucciones define un nuevo programa cuyo listado es 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
```

Para este último programa es muy importante que te des cuenta de que el desarrollo de las cuatro instrucciones nuevas y del programa nuevo 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 este nuevo programa, además de los anteriores. La interacción con el programa debería incluir la 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`. El *main* 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 subdirectorio `'c++'` y `'java'`, cada uno de ellos conteniendo el código 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 *.cc* y generar el ejecutable. No deben incluir ficheros objeto (*.o) o ejecutables en el caso de C++, ni ficheros de clases compiladas (*.class) de Java. Tampoco otros ficheros generados por el entorno de desarrollo (subdirectorio `.vscode`, por ejemplo).

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 hecha 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
```

En caso de no compilar siguiendo estas instrucciones, el resultado de la evaluación de la práctica será de 0. No se deben utilizar paquetes ni librerías ni ninguna infraestructura adicional fuera de las librerías estándar de los correspondientes lenguajes.
Todos los archivos de código fuente 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 por parejas. En este caso sólo uno de los dos estudiantes deberá hacer la entrega.

- `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 que te pedimos: ningún fichero ejecutable o de objeto, ni ningún otro fichero adicional. La entrega se hará en la tarea correspondiente a través de la plataforma Moodle:
`http:://moodle.unizar.es`