Departamento de
Informática e
Ingeniería de Sistemas
Universidad Zaragoza
1542

Práctica 1

Tecnología de Programación

Grado en Ingeniería Informática

Objetivos

- Aprender los aspectos básicos de la sintaxis en programación orientada a objetos mediante la implementación de tipos abstractos de datos sencillos.
- Adquirir la primera experiencia con polimorfismo, tanto por inclusión (herencia) como paramétrico (programación genérica).
- Comprender similitudes y diferencias en la implementación del paradigma orientado a objetos entre los lenguajes C++ y Java, así como sus implicaciones a nivel de gestión de memoria dinámica.

1. TAD en C++

En este apartado diseñarás en C++ el tipo abstracto de datos `static_stack` (pila estática). Para ello te proporcionamos como material adjunto a esta práctica los siguientes ficheros:

- `static-stack-struct.h`
  El tipo de datos `static_stack` definido mediante *`structs`* de C++ y funciones (quizá te resulte similar a asignaturas anteriores). Tiene un iterador que se recorre la estructura de datos **en orden inverso**, desde el último elemento introducido hasta el primero.
- `main-static-stack-struct.cc`
  Programa principal que utiliza el tipo de datos definido en el fichero anterior para guardar y mostrar una secuencia de enteros por pantalla.
- `static-stack.h`
  Es el esqueleto de la definición del tipo de datos pila definido mediante clases y métodos en C++. Deberás rellenar los huecos en el código de este fichero especificados mediante comentarios que empiezan con `TODO:`.
- `main-static-stack.cc`
  El programa principal que utiliza el tipo de datos definido en el fichero anterior. Debería compilar y funcionar correctamente (de forma idéntica a `main-static-stack-struct.cc`) una vez hayas implementado los métodos en el fichero anterior.

- `Makefile`
  Fichero de configuración para ejecutar `make` y compilar todo lo anterior.

Deberás implementar una serie de métodos en `static-stack.h` para que se comporte de forma equivalente a las funciones implementadas en `static-stack-struct.h`. Los métodos que tienes que implementar están marcados con un comentario que empieza con `TODO:` seguido de instrucciones específicas en cada caso.

La implementación del TAD pila mediante una **clase con métodos** presenta una serie de diferencias frente a la inicial mediante un **struct y funciones** que es importante tener en cuenta:

- No existe función `init`. Casi la totalidad de los lenguajes orientados a objetos permiten la definición de un constructor, que hace las veces de inicializador, que tiene el mismo nombre que la clase y que se ejecuta automáticamente siempre que se crea en memoria el objeto correspondiente.
- Al contrario que en las funciones (en la que el parámetro que representa al TAD se declara explícitamente) en una clase los métodos tienen un parámetro implícito `this`, que es un puntero que apunta al propio objeto y que permite acceder a sus atributos y métodos. En la mayoría de los casos se puede omitir.
- La implementación de los iteradores es diferente: el iterador (`iterator`) es una clase definida dentro de la clase global que contiene los atributos y métodos exclusivos de los iteradores. Al separar el iterador de la propia estructura de datos, se pueden tener varios iteradores recorriéndo simultáneamente la estructura de datos.
- Para seguir el estándar definido por la *Standard Template Library* (STL) de C++, los iteradores tienen que tener definidos ciertos métodos de forma particular: la inicialización del iterador es un constructor, el avance y el acceso al iterador están en métodos separados con nombres específicos y la comprobación de finalización se hace comparando iteradores al inicio y al final de la estructura de datos (ver el esqueleto que proporcionamos para conocer más detalles). Esto permite recorrer la estructura de datos con el bucle *foreach* específico del lenguaje, como se puede ver en el programa principal.

2. TAD en Java

Como primera aproximación al lenguaje Java, vamos a diseñar el mismo tipo abstracto de datos que en el apartado anterior pero en el lenguaje Java. De nuevo, te proporcionamos material:

- `StaticStack.java`
  Es el esqueleto de la definición del tipo de datos, que deberás rellenar.
- `MainStaticStack.java`
  Es el programa principal que prueba el tipo de datos definido en la clase anterior.

Deberás implementar una serie de métodos en `StaticStack.java` para que se comporte de forma equivalente a la estructura de datos definida mediante una clase en C++ en el apartado anterior. El código a rellenar está marcado con comentarios que empiezan con *`TODO:`* seguido de instrucciones específicas en cada caso.

Dado que es teóricamente tu primer contacto con el lenguaje Java, te enumeramos las diferencias principales con respecto a C++ que te serán útiles aquí:

- Java es un lenguaje puramente **orientado a objetos**. Esto implica, entre otras cosas, que no existe el concepto de función: todo son métodos que pertenecen a alguna clase.
- Java es un lenguaje **compilado** pero que se ejecuta mediante una **máquina virtual**. Para compilar y ejecutar el ejemplo que te proporcionamos deberás utilizar los siguientes comandos:

```
javac MainStaticStack.java
java MainStaticStack
```

- Todas las clases y vectores en Java son representadas en memoria mediante **punteros**. Esto implica que no hay dos operadores `.` y `->`, sino que únicamente se utiliza el operador `.`. Afortunadamente, en Java hay recolección automática de basura, así que no te tienes que preocupar de la gestión de la memoria dinámica.
- Para los **iteradores**, utilizamos en el esqueleto que te proporcionamos la estructura de la biblioteca estándar de Java. Esto hace cierto uso de la herencia (palabra clave `implements`), mecanismo que es posible que todavía no hayamos visto en clase.

Sorprendentemente, hay pocas diferencias más, aparte de alguna puramente sintáctica.

3. Pila dinámica

La implementación de la pila que has hecho hasta ahora es estática. En C++ es puramente estática (se elige el tamaño en tiempo de compilación y los elementos de la pila se guardan en la pila de activación del bloque en el que se usa la pila) mientras en en Java es de tamaño estático (se fija en compilación) aunque la reserva de espacio se haga en tiempo de ejecución. Esto tiene ciertas ventajas, como la contigüidad en memoria (que favorece a los aciertos en cache) o la no reserva y destrucción de memoria por cada elemento (mayor rapidez de inserción y borrado) pero tiene la desventaja de tener un límite de tamaño establecido en compilación. Como contrapartida a esta estructura, vamos a diseñar una agrupación dinámica (tanto en C++ como en Java) en la que, al introducir cada elemento, se añada un nodo a la estructura de forma dinámica, sin límite de tamaño, salvo el establecido por la propia memoria de la máquina.

Genera dos clases nuevas que representen el mismo TAD pero cambiando a una implementación dinámica. Mantén el interfaz de todos los métodos públicos (tanto de clase como de iterador) cambiando la implementación de los métodos y los atributos:

- Para **C++**, llama a la clase `dynamic_stack` y guárdala en un fichero llamado **dynamic-stack.h**.
  Para probar la nueva clase, puedes usar el programa `main-dynamic-stack.cc` que te damos. Añade al `Makefile` la información necesaria para que compile el programa `main-dynamic-stack` además de los anteriores.

- Para **Java**, llama a la clase `DynamicStack` y guárdala en un fichero llamado **DynamicStack.java**. Para probar la nueva clase, puedes usar el programa `MainDynamicStack.java` que te damos.

Deberás encargarte de gestionar la memoria dinámica (la creación y destrucción, en su caso, de los nodos). Para ello la clase (tanto en C++ como en Java) deberá tener una clase interna (al estilo del iterador) que represente un `Nodo`. Estos nodos se crearán al añadir un elemento y se destruirán cuando se borre un elemento (por ejemplo el último), o cuando el programa acabe. En la liberación / destrucción de la memoria dinámica es donde notarás la mayor diferencia en ambos lenguajes. En Java no es necesario preocuparse de eso (la propia máquina virtual se encarga). Sin embargo, en C++, deberás definir un destructor tanto para la clase `dynamic_stack` como, si lo ves necesario, para la clase interna que represente un nodo. Además, deberás liberar específicamente la memoria al eliminar un elmento de la pila.

4. Polimorfismo

En este punto tienes definidas dos clases (tanto en C++ como en Java) que representan el mismo tipo abstracto de datos con diferentes implementaciones, una estática y otra dinámica. Ahora deberás conseguir generar código que funcione de forma equivalente con ambos tipos de datos. Para ello te proporcionamos archivos que contienen un programa principal (`main.cc` y `Main.java`, en cada caso), cada uno de ellos con dos *`TODO:`* que deberás resolver.

Te proporcionamos (en cada caso) la implementación de dos funciones (métodos estáticos, en Java), pero no te proporcionamos la cabecera de dichas funciones (métodos). Deberás crear la cabecera correcta que permita que la función (método) funcione con las dos clases que has desarrollado de forma polimórfica.

Aquí es, de nuevo, donde la filosofía de Java y de C++ difiere:

- En **C++**, el mecanismo para lograr este tipo de polimorfismo es la programación genérica. Deberás conseguir que las funciones correspondientes sean genéricas y acepten ambos tipos de datos (pila estática y pila dinámica).

- En **Java**, el mecanismo para lograr este tipo de polimorfismo es la herencia. Deberás diseñar un interfaz común a ambas implementaciones de la pila que permita el polimorfismo que buscas (llama a esta nueva clase o interfaz como quieras).

En ningún caso modifiques el resto del código de los archivos `main.cc` y `Main.java`, ni el programa principal ni la implementación de los correspondientes métodos y funciones. Sólo deberás

modificar **las cabeceras** de los correspondientes métodos y funciones para conseguir el objetivo propuesto, y añadir las reglas correspondientes al `Makefile` en el caso de C++.

Entrega

Los archivos de código fuente de la práctica deberán estar separados en dos subdirectorios diferentes, con la siguiente estructura (XXXXXX es tu NIP, o los NIPs de la pareja de prácticas, ver más adelante):

```
practica1_XXXXXX
     \---- c++
          \---- static-stack.h
          \---- dynamic-stack.h
          \---- main.cc
          \---- Makefile
          \---- ...
     \---- java
          \---- StaticStack.java
          \---- DynamicStack.java
          \---- Main.java
          \---- ...
```

Cada subdirectorio podrá incluir otros archivos de código fuente si se considera necesario.

El programa en C++ deberá ser compilable y ejecutable con los archivos que has entregado desde la subcarpeta `c++` mediante los siguientes comandos:

```
make
./main
```

El programa en Java deberá ser compilable y ejecutable con los archivos que has entregado desde la subcarpeta `java` 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 propios lenguajes.

Todos los archivos de código fuente solicitados en este guión deberán ser comprimidos en un único archivo `ZIP` con el siguiente nombre:

- `practica1_<nip>.zip` (donde `<nip>` es el NIP de 6 dígitos del estudiante involucrado) si el trabajo ha sido realizado de forma individual.
- `practica1_<nip1>_<nip2>.zip` (donde `<nip1>` y `<nip2>` son los NIPs de 6 dígitos 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.

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.